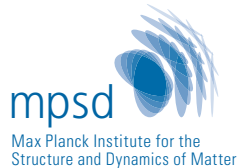


Python for Computational Science




Hans Fangohr

Max Planck Institute for the Structure and Dynamics of Matter
Hamburg (Germany)

University of Southampton (UK)
January 27, 2023



`hans.fangohr@mpsd.mpg.de`
<https://fangohr.github.io>
@ProfCompMod@fosstodon.org 

Outline

1. Python for Computational Science
2. Python prompt
3. Functions
4. About Python
5. Coding style
6. Conditionals, if-else
7. Sequences
8. Loops
9. Some things revisited
10. Reading and Writing files
11. Exceptions
12. Printing

13. Higher Order Functions
14. Modules
15. Default arguments
16. Namespaces
17. Editors and IDEs
18. List comprehension
19. Dictionaries
20. Recursion
21. Common Computational Tasks
22. Root finding
23. Derivatives
24. Numpy
25. Higher Order Functions 2: Functional tools
26. Numerical Integration
27. Numpy usage examples

28. Closures
29. Scientific Python
30. FIFO example and Object Oriented Programming (OOP)
31. Environments and Python Package Index
32. ODEs
33. Sympy
34. Testing
35. Object Oriented Programming
36. Pandas
37. Practical computational science recommendations
38. What to learn next?
39. Typing
40. Useful tools

Python for Computational Science

- use of computers to support research and operation in science, engineering, industry and services
- applications include
 - computer simulations
 - analysis of data / data science / data analytics
 - virtual design optimisation
 - symbolic mathematics
 - artificial intelligence/machine learning

- Molecular Dynamics (<https://physics.weber.edu/schroeder/software/demos/MDv0.html>)
- Imaging at European X-Ray Free Electron Laser (European XFEL)
 - large data sets (data creation up to 1000TB per week)
 - Computational modelling to extract structure of sample

Is computational science important?

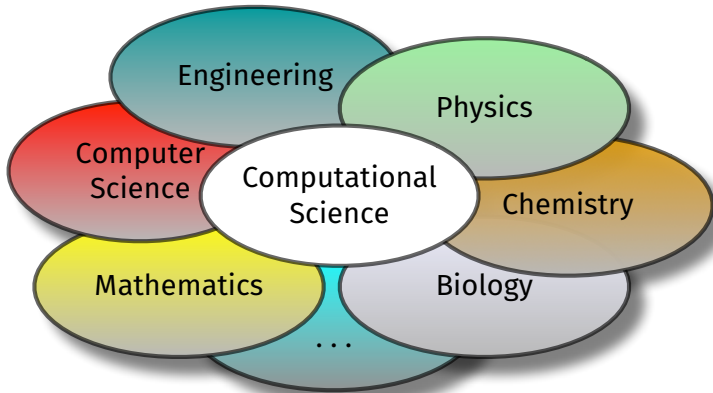
From a study of the Software Sustainability Institute (UK):

- 92% of academics use research software
- 69% say that their research would not be practical without it
- 56% develop their own software

[1] <https://www.software.ac.uk/blog/>

[2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers](https://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers)

Computational Science — a new domain



Computational science (and data science):
an *enabling methodology*, like literacy and mathematics

Computational Science — a new domain

- Computational Science is not Computer Science
- specific skill set required: application domain knowledge *and* computational science
- often scientists who learn the computational side
- no clear career path: neither scientist nor software engineer
- growing movement to establish such roles in academia: Research Software Engineer
 - <https://www.software.ac.uk>
 - <https://www.de-rse.org>
- “better software, better research”



This course: Introduction to Python for Computational Science

- introduces the foundations of computational science and data science
- Python programming language
- focus on parts of the Python programming language relevant to computational science
- computational science methodology
- research software engineering
- enable self-directed learning in the future

This course: Why Python?

- Python is relatively easy to learn [1]
- high efficiency: a few lines of code achieve a lot of computation
- growing use in (open source) academia and industry, thus
- many relevant libraries available
- minimises the time of the programmer
- but: (naive) Python in general much slower execution than compiled languages (such as Fortran, C, C++, Rust, ...).

[1] [https:](https://link.springer.com/chapter/10.1007/978-3-540-25944-2_157)

[//link.springer.com/chapter/10.1007/978-3-540-25944-2_157](https://link.springer.com/chapter/10.1007/978-3-540-25944-2_157)

This course: learning methods

- daily lectures
- daily laboratory sessions (think computer laboratory)
 - opportunity to start and complete self-paced exercises, and to ask for any other clarification.
- automatic feedback on submitted exercises
- teaching materials and lectures are designed to support practical exercises

Source of information:

<http://www.desy.de/~fangohr/teaching/py4cs2023>

- time table
- laboratory exercises
- pdf files of these slides (may change)
- additional textbook
- further materials

Remote learning brings extra challenges! Let's try to make this informal and interactive (Zoom & Zulip)

Python prompt

The Python prompt

- Spyder (or IDLE, or `python` or `python.exe` from shell/Terminal/MS-Dos prompt, or IPython)
- Python prompt waits for input:
`>>>`
- Interactive Python prompt waits for input:
`In [1]:`
- Read, Evaluate, Print, Loop → REPL

Hello World program

Standard greeting:

```
print("Hello World")
```

Entered interactively in Python prompt:

```
>>> print("Hello World")
```

```
Hello World
```

Or in IPython prompt:

```
In [1]: print("Hello world")
```

```
Hello world
```

A calculator

```
>>> 2 + 3
```

```
5
```

```
>>> 42 - 15.3
```

```
26.7
```

```
>>> 100 * 11
```

```
1100
```

```
>>> 2400 / 20
```

```
120
```

```
>>> 2 ** 3
```

```
# 2 to the power of 3
```

```
8
```

```
>>> 9 ** 0.5
```

```
# sqrt of 9
```

```
3.0
```

Create variables through assignment

```
>>> a = 10
```

```
>>> b = 20
```

```
>>> a
```

```
10
```

```
>>> b
```

```
20
```

```
>>> a + b
```

```
30
```

```
>>> ab2 = (a + b) / 2
```

```
>>> ab2
```

```
15
```

Important data types / type()

```
>>> a = 1
>>> type(a)
<class int>           # integer

>>> b = 1.0
>>> type(b)
<class float>        # float

>>> c = '1.0'
>>> type(c)
<class str>          # string

>>> d = 1 + 3j
>>> type(d)
<class complex>     # complex number
```

Summary useful commands (introspection)

- `print(x)` to display the object `x`
- `type(x)` to determine the type of object `x`
- `help(x)` to obtain the documentation string
- `dir(x)` to display the methods and members of object `x`, or the current name space (`dir()`).

Example:

```
>>> help("abs")  
Help on built-in function abs:
```

```
abs(...)  
    abs(number) -> number
```

Return the absolute value of the argument.

Interactive documentation, introspection

```
>>> word = 'test'
>>> print(word)
test
>>> type(word)
<class str>
>>> dir(word)
['__add__', '__class__', '__contains__', ...,
 '__doc__', ..., 'capitalize', <snip>,
 'endswith', ..., 'upper', 'zfill']
>>> word.upper()
'TEST'
>>> word.capitalize()
'Test'
>>> word.endswith('st')
True
>>> word.endswith('a')
False
```

Functions

Example 1:

```
def mysum(a, b):  
    return a + b
```

```
# main program starts here  
print("The sum of 3 and 4 is", mysum(3, 4))
```


Functions should be documented

```
def mysum(a, b):  
    """Return the sum of parameters a and b."""  
    return a + b
```

```
# main program starts here  
print("The sum of 3 and 4 is", mysum(3, 4))
```

Can now use the help function for our new function:

```
>>> help(mysum)  
Help on function mysum in module __main__:  
  
mysum(a, b)  
    Return the sum of parameters a and b.
```

Function documentation strings

```
def mysum(a, b):  
    """Return the sum of parameters a and b."""  
    return a + b
```

Essential information for documentation string:

- What inputs does the function expect?
- What does the function do?
- What does it return

Desirable:

- Examples
- Notes on algorithm (if relevant)
- exceptions that might be raised
- [Author, date, contact details: not needed if version control is used]

LAB1

Advanced: Recommendations for documentation string style are [numpydoc style](#) or [PEP257 docstring conventions](#).

Function documentation string example 1

```
def mysum(a, b):
    """Return the sum of parameters a and b.

    Parameters
    -----
    a : numeric
        first input
    b : numeric
        second input

    Returns
    -----
    a+b : numeric
        returns the sum (using the + operator) of a and b. The return type will
        depend on the types of `a` and `b`, and what the plus operator returns.

    Examples
    -----
    >>> mysum(10, 20)
    30
    >>> mysum(1.5, -4)
    -2.5

    Notes
    -----
    History: example first created 2002, last modified 2013
    Hans Fangohr, fangohr@soton.ac.uk,
    """
    return a + b
```

Function documentation string example 2

```
def factorial(n):
    """Compute the factorial.

    Parameters
    -----
    n : int
        Natural number `n` > 0 for which the factorial is computed.

    Returns
    -----
    n! : int
        Returns n * (n-1) * (n-2) * ... * 2 * 1

    Examples
    -----
    >>> factorial(1)
    1
    >>> factorial(3)
    6
    >>> factorial(10)
    3628800
    >>> factorial(20)
    2432902008176640000
    """
    assert n > 0

    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Function terminology

```
x = -1.5
```

```
y = abs(x)
```

- x is the *argument* given to the function (also called *input* or *parameter*)
- y is the *return value* (the result of the function's computation)
- Functions may expect zero, one or more arguments
- Not all functions (seem to) return a value. (If no **return** keyword is used, the special object **None** is returned.)

Function example

```
def plus42(n):  
    """Add 42 to n and return""" # docstring  
    l = n + 42                    # body of  
                                  # function  
    return l  
  
a = 8  
b = plus42(a)    # not part of function definition
```

After execution, **b** carries the value 50 (and **a** = 8).

Summary functions

- Functions provide (black boxes of) functionality: crucial building blocks that hide complexity
- interaction (input, output) through input arguments and return values (*printing* and *returning* values is not the same!)
- docstring provides the specification (contract) of the function's input, output and behaviour
- a function should (normally) not modify input arguments (watch out for lists, dicts, more complex data structures as input arguments)

Functions printing vs returning values

Given the following two function definitions:

```
def print42():  
    print(42)
```

```
def return42():  
    return 42
```

we use the Python prompt to explore the difference:


```
>>> b = return42() # return 42, is assigned
>>> print(b)      # to b
42

>>> a = print42() # return None, and
42               # print 42 to screen
>>> print(a)
None             # special object None
```

If we use IPython, it shows whether a function returns something (i.e. not None) through the `Out []` token:

```
In [1]: return42()
```

```
Out[1]: 42                # Return value of 42
```

```
In [2]: print42()
```

```
42                # No 'Out [ ]', so no  
                  # returned value
```


About Python

What is Python?

- High level programming language
- interpreted
- supports three main programming styles (imperative=procedural, object-oriented, functional)
- General purpose tool, yet good for numeric work with extension libraries

Availability

- Python is free
- Python is platform independent (works on Windows, Linux/Unix, Mac OS, ...)
- Python is open source

There is lots of documentation that you should learn to use:

- Teaching materials on website, including these slides and a **text-book** like document
 - Online documentation, for example
 - Python home page (<http://www.python.org>)
 - **Matplotlib** (publication figures)
 - **Numpy** (fast vectors and matrices, (NUMerical PYthon))
 - **SciPy** (scientific algorithms, `solve_ivp`)
 - **SymPy** (Symbolic calculation)
- interactive documentation

Which Python version

- There are currently two versions of Python:
 - Python 2.7 and
 - Python 3.x
- We will use version 3.9 or later
- Python 2.x and 3.x are incompatible although the changes only affect very few commands.
- Write new programs in Python 3.
- You may have to read / work with Python 2 code at some point.

The math module (`import math`)

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.pi
3.141592653589793
>>> dir(math)          #attributes of 'math' object
['__doc__', '__file__', < snip >
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'e', 'erf',
'exp', <snip>, 'sqrt', 'tan', 'tanh', 'trunc']

>>> help(math.sqrt)    # ask for help on sqrt
sqrt(...)
    sqrt(x)
    Return the square root of x.
```


Name spaces and modules

Three (good) options to access a module:

1. use the full name:

```
import math
print(math.sin(0.5))
```

2. use some abbreviation

```
import math as m
print(m.sin(0.5))
print(m.pi)
```

3. import all objects we need explicitly

```
from math import sin, pi
print(sin(0.5))
print(pi)
```

Python 2: Integer division

Dividing two integers in Python 1 and 2 returns an integer:

```
>>> 1 / 2
0          # might have expected 0.5, not 0
```

We find the same behaviour in Java, C, Fortran, and many other programming languages.

Solutions:

- change (at least) one of the integer numbers into a floating point number (i.e. $1 \rightarrow 1.0$).

```
>>> 1.0 / 2
0.5
```

- Or use `float` function to convert variable to float

```
>>> a = 1
>>> b = 2
>>> 1 / float(b)
0.5
```

- Or make use of Python's future division:

```
>>> from __future__ import division
>>> 1 / 2
0.5
```

Python 3: Integer division

In Python 3:

```
>>> 1 / 2  
0.5
```

Dividing 2 integers returns a float:

```
>>> 4 / 2  
2.0  
>>> type(4 / 2)  
<class float>
```

If we want integer division (i.e. an operation that returns an integer, and/or which replicates the default behaviour of Python 2), we use `//`:

```
>>> 1 // 2  
0
```

Coding style

- Python programs *must* follow Python syntax.
- Python programs *should* follow Python style guide, because
 - readability is key (debugging, documentation, team effort)
 - conventions improve effectiveness

Common style guide: PEP8

From <http://www.python.org/dev/peps/pep-0008/>:

- This document gives coding conventions for the Python code [...]
- This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.
- Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.
- One of Guido van Rossum's key insights is that code is *read much more often than it is written*. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.
"Readability counts".

- Indentation: use 4 spaces
- One space around assignment operator (=) operator:
`c = 5` and not `c=5`.
- Spaces around arithmetic operators can vary. Both
`x = 3*a + 4*b` and `x = 3 * a + 4 * b` are okay.
- No space before and after parentheses:
`x = sin(x)` but not `x = sin(x)`
- A space after comma: `range(5, 10)` and not
`range(5,10)`.
- No whitespace at end of line
- No whitespace in empty line

- One or no empty line between statements within function
- Two empty lines between functions
- One import statement per line
- import first standard Python library (such as `math`), then third-party packages (`numpy`, `scipy`, ...), then our own modules
- no spaces around = when used in keyword arguments:
`"Hello World".split(sep=' ')` but not
`"Hello World".split(sep = ' ')`

PEP8 Style Summary

- Try to follow PEP8 guide, in particular for new code.
- Use tools to help us, for example Spyder editor can show PEP8 violations.

Similar tools/plugins are available for other editors.
editors.

- `pycodestyle` program available to check source code from command line (used to be called `pep8` in the past).

To check file `myfile.py` for PEP8 compliance:

```
pycodestyle myfile.py
```

Style conventions for documentation strings

- Python documentation strings (pydoc) conventions:
 - **PEP257 docstring style** (from 2001), basis for both
 - **numpydoc style** (science) and
 - **Google pydoc style**
- Examples on slide 26 and 27 are compatible with all conventions
- Editors can highlight deviations
- Program to check documentation string style compliance in file `myfile.py`:
 - `pydocstyle --convention=pep257 myfile.py`
 - `pydocstyle --convention=numpy myfile.py`
 - `pydocstyle --convention=google myfile.py`

Conditionals, if-else

Truth values

The python values `True` and `False` are special inbuilt objects:

```
>>> a = True
>>> print(a)
True
>>> type(a)
<class bool>
>>> b = False
>>> print(b)
False
>>> type(b)
<class bool>
```

We can operate with these two logical values using boolean logic, for example the logical and operation (and):

```
>>> True and True           # logical and operation
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

There is also logical or (or) and the negation (not):

```
>>> True or False
```

```
True
```

```
>>> not True
```

```
False
```

```
>>> not False
```

```
True
```

```
>>> True and not False
```

```
True
```


In computer code, we often need to evaluate some expression that is either true or false (sometimes called a “predicate”).
For example:

```
>>> x = 30          # assign 30 to x
>>> x >= 30         # is x greater than or equal to 30?
True
>>> x > 15          # is x greater than 15
True
>>> x > 30          # is x greater than 30?
False
>>> x == 30         # is x the same as 30?
True
>>> not x == 42     # is x not the same as 42?
True
>>> x != 42         # is x not the same as 42?
True
```

The `if-else` command allows to branch the execution path depending on a condition. For example:

```
>>> x = 30                # assign 30 to x
>>> if x > 30:           # predicate: is x > 30
...     print("Yes")     # if True, do this
... else:
...     print("No")      # if False, do this
...
No
```

The general structure of the `if-else` statement is

```
if A:  
    B  
else:  
    C
```

where `A` is the predicate.

- If `A` evaluates to `True`, then all commands `B` are carried out (and `C` is skipped).
- If `A` evaluates to `False`, then all commands `C` are carried out (and `B` is skipped).
- `if` and `else` are Python keywords.

`A` and `B` can each consist of multiple lines, and are grouped through indentation as usual in Python.

if-else example

```
def slength1(s):  
    """Returns a string describing the  
    length of the sequence s"""  
    if len(s) > 10:  
        ans = 'very long'  
    else:  
        ans = 'normal'  
  
    return ans
```

```
>>> slength1("Hello")  
'normal'  
>>> slength1("HelloHello")  
'normal'  
>>> slength1("Hello again")  
'very long'
```

if-elif-else example

If more cases need to be distinguished, we can use the keyword `elif` (standing for ELse IF) as many times as desired:

```
def slength2(s):  
    if len(s) == 0:  
        ans = 'empty'  
    elif len(s) > 10:  
        ans = 'very long'  
    elif len(s) > 7:  
        ans = 'normal'  
    else:  
        ans = 'short'  
  
    return ans
```

```
>>> slength2("")
'empty'
>>> slength2("Good Morning")
'very long'
>>> slength2("Greetings")
'normal'
>>> slength2("Hi")
'short'
```

LAB2

Sequences

Different types of sequences

- strings
- lists (mutable)
- tuples (immutable)
- arrays (mutable, part of numpy)

They share common behaviour.

Strings

```
>>> a = "Hello World"
>>> type(a)
<class str>
>>> len(a)
11
>>> print(a)
Hello World
```

Different possibilities to limit strings:

```
'A string'
"Another string"
"A string with a ' in the middle"
"""A string with triple quotes can
extend over several
lines"""
```

Strings 2 (exercise)

- Define a, b and c at the Python prompt:

```
>>> a = "One"
```

```
>>> b = "Two"
```

```
>>> c = "Three"
```

- Exercise: What do the following expressions evaluate to?

1. `d = a + b + c`

2. `5 * d`

3. `d[0]`, `d[1]`, `d[2]` (indexing)

4. `d[-1]`

5. `d[4:]` (slicing)

Strings 3 (exercise)

```
>>> s="""My first look at Python was an  
... accident, and I didn't much like what  
... I saw at the time."""
```

For the string `s`:

- count the number of (i) letters 'e' and (ii) substrings 'an'
- replace all letters 'a' with '0'
- make all letters uppercase
- make all capital letters lowercase, and all lower case letters to capitals

```
[] # the empty list
[42] # a 1-element list
[5, 'hello', 17.3] # a 3-element list
[[1, 2], [3, 4], [5, 6]] # a list of lists
```

- Lists store an ordered sequence of Python objects
- Access through index (and slicing) as for strings.
- use `help()`, often used list methods is `append()`

(In general computer science terminology, vector or array might be better name as the actual implementation is not a linked list, but direct $\mathcal{O}(1)$ access through the index is possible.)

Example program: using lists

```
>>> a = []                # creates a list
>>> a.append('dog')      # appends string 'dog'
>>> a.append('cat')      # ...
>>> a.append('mouse')
>>> print(a)
['dog', 'cat', 'mouse']
>>> print(a[0])          # access first element
dog                       # (with index 0)
>>> print(a[1])          # ...
cat
>>> print(a[2])
mouse
>>> print(a[-1])         # access last element
mouse
>>> print(a[-2])        # second last
cat
```

Example program: lists containing a list

```
>>> a = ['dog', 'cat', 'mouse', [1, 10, 100, 1000]]
>>> a
['dog', 'cat', 'mouse', [1, 10, 100, 1000]]
>>> a[0]
dog
>>> a[3]
[1, 10, 100, 1000]
>>> max(a[3])
1000
>>> min(a[3])
1
>>> a[3][0]
1
>>> a[3][1]
10
>>> a[3][3]
1000
```

Sequences – more examples

```
>>> a = "hello world"
>>> a[4]
'o'
>>> a[4:7]
'o w'
>>> len(a)
11
>>> 'd' in a
True
>>> 'x' in a
False
>>> a + a
'hello worldhello world'
>>> 3 * a
'hello worldhello worldhello world'
```

Tuples

- tuples are very similar to lists
- tuples are *immutable* (unchangeable) whereas lists are *mutable* (changeable)
- tuples are usually written using parentheses (\leftrightarrow “round brackets”):

```
>>> t = (3, 4, 50)    # t for Tuple
```

```
>>> t
```

```
(3, 4, 50)
```

```
>>> type(t)
```

```
<class tuple>
```

```
>>> l = [3, 4, 50]    # compare with l for List
```



```
>>> l
[3, 4, 50]
>>> type(l)
<class list>
```

Tuples are defined by comma

- tuples are defined by the comma (!), not the parenthesis

```
>>> a = 10, 20, 30
```

```
>>> type(a)
```

```
<class tuple>
```

- the parentheses are usually optional (but should be written anyway):

```
>>> a = (10, 20, 30)
```

```
>>> type(a)
```

```
<class tuple>
```

- normal indexing and slicing (because tuple is a sequence)

```
>>> t[1]
```

```
4
```

```
>>> t[:-1]
```

```
(3, 4)
```

Why do we need tuples (in addition to lists)?

1. use tuples if you want to make sure that a set of objects doesn't change.
2. Using tuples, we can assign several variables in one line (known as *tuple packing* and *unpacking*)

```
x, y, z = 0, 0, 1
```

This allows “instantaneous swap” of values:

```
a, b = b, a
```

Strictly: “tuple packing” on right hand side and “sequence unpacking” on left.

3. functions return tuples if they return more than one object

```
def f(x):  
    return x**2, x**3
```

```
a, b = f(x)
```

4. tuples can be used as keys for dictionaries as they are immutable

(Im)mutables

- Strings — like tuples — are immutable:

```
>>> a = 'hello world'           # String example
>>> a[3] = 'x'
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

```
TypeError: object does not support item assignment
```

- strings can only be 'changed' by creating a new string, for example:

```
>>> a = a[0:3] + 'x' + a[4:]
>>> a
'helxo world'
```

Summary sequences

- lists, strings and tuples (and arrays) are sequences.
- sequences share the following operations

<code>a[i]</code>	returns element with index i of <code>a</code>
<code>a[i:j]</code>	returns elements i up to $j - 1$
<code>len(a)</code>	returns number of elements in sequence
<code>min(a)</code>	returns smallest value in sequence
<code>max(a)</code>	returns largest value in sequence
<code>x in a</code>	returns <code>True</code> if <code>x</code> is element in <code>a</code>
<code>a + b</code>	concatenates <code>a</code> and <code>b</code>
<code>n * a</code>	creates <code>n</code> copies of sequence <code>a</code>

In the table above, `a` and `b` are sequences, `i`, `j` and `n` are integers, `x` is an element.

Conversions

- We can convert any sequence into a tuple using the `tuple` function:

```
>>> tuple([1, 4, "dog"])  
(1, 4, 'dog')
```

- Similarly, the `list` function, converts sequences into lists:

```
>>> list((10, 20, 30))  
[10, 20, 30]
```

- *Looking ahead* to iterators, we note that `list` and `tuple` can also convert from iterators:

```
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

And if you ever need to create an iterator from a sequence, the `iter` function can this:

```
>>> iter([1, 2, 3])  
<list_iterator object at 0x1013f1fd0>
```


Loops

Introduction loops

Computers are good at repeating tasks (often the same task for many different sets of data).

Loops are the way to execute the same (or very similar) tasks repeatedly (“in a loop”).

Python provides the “for loop” and the “while loop”.

Example program: for-loop

```
animals = ['dog', 'cat', 'mouse']  
  
for animal in animals:  
    print(f"This is the {animal}!")
```

produces

```
This is the dog!
```

```
This is the cat!
```

```
This is the mouse!
```

The for-loop *iterates* through the sequence `animals` and assigns the values in the sequence subsequently to the name `animal`.

Iterating over integers

Often we need to iterate over a sequence of integers:

```
for i in [0, 1, 2, 3, 4, 5]:  
    print(f"the square of {i} is {i**2}")
```

produces

```
the square of 0 is 0  
the square of 1 is 1  
the square of 2 is 4  
the square of 3 is 9  
the square of 4 is 16  
the square of 5 is 25
```

Iterating over integers with range

The `range(n)` object is used to iterate over a sequence of increasing integer values up to (but not including) `n`:

```
for i in range(6):  
    print(f"the square of {i} is {i**2}")
```

produces

```
the square of 0 is 0  
the square of 1 is 1  
the square of 2 is 4  
the square of 3 is 9  
the square of 4 is 16  
the square of 5 is 25
```

The range object

- `range` is used to iterate over integer sequences
- (Advanced:) `range` has its own type:

```
>>> type(range(6))  
<class range>
```

- We can use the range object in for loops:

```
>>> for i in range(3):  
...     print(f"i={i}")  
i=0  
i=1  
i=2
```

- We can convert it to a list:

```
>>> list(range(6))  
[0, 1, 2, 3, 4, 5]
```

- This conversion to list is useful to understand what sequences the range object would provide if used in a for loop:

```
>>> list(range(6))  
[0, 1, 2, 3, 4, 5]  
>>> list(range(0, 6))  
[0, 1, 2, 3, 4, 5]  
>>> list(range(3, 6))  
[3, 4, 5]  
>>> list(range(-3, 0))  
[-3, -2, -1]
```

Summary range

range

`range([start,] stop [,step])` iterates over integers from `start` to `stop` (*but not including stop*) in steps of `step`.

`start` defaults to 0 and `step` defaults to 1.

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 4))
[]                                # no iterations
```

range objects are lazy sequences (**Python range is not an iterator**)

Iterating over sequences with for-loop

- for loop iterates over iterables.
- Sequences are iterable.

```
for i in [0, 3, 4, 19]:  
    print(i)
```

```
for animal in ['dog', 'cat', 'mouse']:  
    print(animal)
```

```
for letter in "Hello World":  
    print(letter)                # strings are  
                                # sequences
```

```
for i in range(5):  
    print(i)                    # range objects  
                                # are sequences
```

- Example 1 (if-then-else)

```
a = 42
if a > 0:
    print("a is positive")
else:
    print("a is negative or zero")
```

Another iteration example

This example generates a list of numbers often used in hotels to label floors ([more info](#))

```
def skip13(a, b):
    """Given ints a and b, return
    list of ints from a to b without 13"""
    result = []
    for k in range(a, b):
        if k == 13:
            pass                # do nothing
        else:
            result.append(k)
    return result
```

Another iteration example 2

This example generates a list of numbers often used in hotels to label floors ([more info](#))

```
def skip13(a, b):
    result = []
    for k in range(a, b):
        if k == 13:
            continue # jump to next iteration

        result.append(k)
    return result
```

Exercise range_double

Write a function `range_double(n)` that generates a list of numbers similar to `list(range(n))`. In contrast to `list(range(n))`, each value in the list should be multiplied by 2. For example:

```
>>> range_double(4)
[0, 2, 4, 6]
>>> range_double(10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

For comparison the behaviour of `range`:

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For loop summary

- for-loop to iterate over sequences
- can use **range** to generate sequences of integers
- special keywords:
 - **continue** - skip remainder of body of statements and continue with next iteration
 - **break** - leave for-loop immediately
- Advanced:
 - can iterate over any *iterable*
 - we can create our own iterables
 - See summary [Socratica on Iterators, Iterables, and Itertools](#)

Exercise: First In First Out (FIFO) queue

Write a *First-In-First-Out* queue implementation, with functions:

- **add(name)** to add a customer with name **name** (call this when a new customer arrives)
- **next()** to be called when the next customer will be served. This function returns the name of the customer
- **show()** to print all names of customers that are currently waiting
- **length()** to return the number of currently waiting customers

Suggest to use a global variable `q` and define this in the first line of the file by assigning an empty list: `q = []`.

While loops

- Reminder: a `for` loop iterates over a given sequence or iterator
- A `while` loop iterates *while a condition is fulfilled*

Example:

```
x = 64
while x > 10:
    x = x // 2
    print(x)
```

produces

32

16

8

Determine ϵ :

```
eps = 1.0
```

```
while eps + 1 > 1:  
    eps = eps / 2.0  
print(f"epsilon is {eps}")
```

Output:

```
epsilon is 1.11022302463e-16
```

Iterables and iterators (advanced)

- an object is *iterable* if the for-loop can iterate over it
- an *iterator* has a `__next()` method, i.e. can be used with `next()`. The iterator is iterable.

```
>>> i = iter(["dog", "cat"])      # create iterator
                                   # from list
```

```
>>> next(i)
```

```
'dog'
```

```
>>> next(i)
```

```
'cat'
```

```
>>> next(i)                       # reached end
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
StopIteration
```

Generators (advanced)

- Generators are functions defined using `yield` instead of `return`
- When called, a generator returns an *object that behaves like an iterator*: it has a `next` method.

```
>>> def squares(n):  
...     for i in range(n):  
...         yield i**2  
...  
>>> s = squares(3)  
>>> next(s)  
0
```

```
>>> next(s)
```

```
1
```

```
>>> next(s)
```

```
4
```

```
>>> next(s)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
StopIteration
```

The execution flow returns at the `yield` keyword (similar to `return`), but the flow continues after the `yield` when the `next` method is called the next time.

A more detailed example demonstrates this:

```
def squares(n):
    print("begin squares()")
    for i in range(n):
        print(f" before yield i={i}")
        yield i**2
        print(f" after yield i={i}")
```

```
>>> g = squares(3)
>>> next(g)
begin squares()
 before yield i= 0
0
>>> next(g)
 after yield i= 0
 before yield i= 1
```

```
1
>>> next(g)
    after yield i= 1
    before yield i= 2
```

```
4
>>> next(g)
    after yield i= 2
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
StopIteration
```

See also [Socratica on Iterators, Iterables, and Itertools](#)

Some things revisited

What are variables?

In Python, variables are references to (or names of) objects.

This is why in the following example, **a** and **b** represent the same list: **a** and **b** are two *different references* to the *same object*:

```
>>> a = [0, 2, 4, 6] # bind name 'a' to list
>>> a                # object [0,2,4,6].
[0, 2, 4, 6]
>>> b = a           # bind name 'b' to the same
>>> b               # list object.
[0, 2, 4, 6]
>>> b[1]           # show second element in list
2                  # object.
>>> b[1] = 10      # modify 2nd element (via b).
>>> b              # show b.
[0, 10, 4, 6]
>>> a              # show a.
[0, 10, 4, 6]
```

- Two objects `a` and `b` are the *same object* if they live in the same place in memory.
- Python provides the `id` function that returns the *identity* of an object. (It is the memory address.)
- We check with `id(a) == id(b)` or `a is b` whether `a` and `b` are the *same object*.
- Two different objects can have the *same value*. We check with `==` See “Equality and identity”, section 3.5

Example 1

```
>>> a = 1
>>> b = 1.0
>>> id(a); id(b)
4298187624           # not in the same place
4298197712           # in memory
>>> a is b           # i.e. not the same objects
False
>>> a == b           # but carry the same value
True
```

Example 2

```
>>> a = [1, 2, 3]
>>> b = a      # b is reference to object of a
>>> a is b     # thus they are the same
True
>>> a == b    # the value is (of course) the same
True
```

Functions – side effect

If we carry out some activity A, and this has an (unexpected) effect on something else, we speak about *side effects*.

Example:

```
def sum(xs):
    s = 0
    for i in range(len(xs)):
        s = s + xs.pop()
    return s

xs = [10, 20, 30]
print(f"xs = {xs};    ", end='')
print(f"sum(xs)={sum(xs)};    ", end='')
print(f"xs = {xs}")
```

Output:

```
xs = [10, 20, 30];    sum(xs)=60;    xs = []
```

Functions - side effect 2

Better ways to compute the sum of a list `xs` (or sequence in general)

- use in-built command `sum(xs)`
- use indices to iterate over list

```
def sum(xs):  
    s=0  
    for i in range(len(xs)):  
        s = s + xs[i]  
    return s
```

- or (better): iterate over list elements directly

```
def sum(xs):  
    s=0  
    for elem in xs  
        s = s + elem  
    return s
```

To print or to return?

- A function that returns the control flow through the `return` keyword, will return the object given after `return`.
- A function that does not use the `return` keyword, returns the special object `None`.
- Generally, functions should return a value
- Generally, functions should not print anything
- Calling functions from the prompt can cause some confusion here: if the function returns a value and the value is not assigned, it will be printed.
- See slide 31.

Reading and Writing files

File input/output

It is a (surprisingly) common task to

- read some input data file
- do some calculation/filtering/processing with the data
- write some output data file with results

Python distinguishes between

- *text* files ('t')
- *binary* files 'b')

If we don't specify the file type, Python assumes we mean text files.

Writing a text file

```
>>> f = open('test.txt', 'wt') # Write Textfile
>>> f.write("first line\nsecond line")
22      # returns number of chars written
>>> f.close()
```

creates a file `test.txt` that reads

```
first line
second line
```

- To write data, we need to open the file with 'w' mode:

```
f = open('test.txt', 'w')
```

By default, Python assumes we mean text files. However, we can be explicit and say that we want to create a Text file for Writing:

```
f = open('test.txt', 'wt')
```

- If the file exists, it will be overridden with an empty file when the open command is executed.
- The file object `f` has a method `f.write` which takes a string as in input argument.
- Must close file at the end of writing process using `f.close()`.

Reading a text file

We create a file object `f` using

```
>>> f = open('test.txt', 'rt') # Read Textfile
```

and have different ways of reading the data:

1. `f.readlines()` returns a list of strings (each being one line)

```
>>> f = open('test.txt', 'rt')
>>> lines = f.readlines()
>>> f.close()
>>> lines
['first line\n', 'second line']
```

2. `f.read()` returns one long string for the whole file

```
>>> f = open('test.txt', 'rt')
>>> data = f.read()
>>> f.close()
>>> data
'first line\nsecond line'
```

3. Use text file `f` as an iterable object: process one line in each iteration (important for large files):

```
>>> f = open('test.txt', 'rt')
>>> for line in f:
...     print(line, end='')
...
first line
```

second line

```
>>> f.close()
```

Opening and *automatic* file closing through context manager

Python provides *context managers* that we use using `with`.
For the file access:

```
>>> with open('test.txt', 'rt') as f:
...     data = f.read()
...
>>> data
'first line\nsecond line'
```

If we use the file context manager, it will close the file automatically (when the control flows leaves the indented block).

Once you are familiar with file access, we recommend you use this method.

Use case: Reading a file, iterating over lines

- Often we want to process line by line. Typical code fragment:

```
f = open('myfile.txt', 'rt')
lines = f.readlines()
f.close()
# some processing of the lines object
```

- It is good practice to close a file as soon as possible.
-

- Better: using the context manager:

```
with open('myfile.txt', 'rt') as f:
    lines = f.readlines()
# some processing of the lines object
```


Splitting a string

- We often need to split a string into smaller parts: use string method `split()`:
(try `help("").split`) at the Python prompt for more info)

Example:

```
>>> c = 'This is my string'
>>> c.split()
['This', 'is', 'my', 'string']
>>> c.split('i')
['Th', 's ', 's my str', 'ng']
```

Exercise: Shopping list

Given a list

bread	1	1.39
tomatoes	6	0.26
milk	3	1.45
coffee	3	2.99

Write program that computes total cost per item, and writes to `shopping_cost.txt`:

bread	1.39
tomatoes	1.56
milk	4.35
coffee	8.97

One solution

One solution is `shopping_cost.py`

```
fin = open('shopping.txt', 'tr')      # INput File
lines = fin.readlines()
fin.close()                          # close file as soon as possible

fout = open('shopping_cost.txt', 'tw') # OUTput File
for line in lines:
    words = line.split()
    itemname = words[0]
    number = int(words[1])
    cost = float(words[2])
    totalcost = number * cost
    fout.write(f"{itemname:20} {totalcost}\n")
fout.close()
```

Exercise

Write function `print_line_sum_of_file(filename)` that reads a file of name `filename` containing numbers separated by spaces, and which computes and prints the sum for each line. A data file might look like

```
1 2 4 67 -34 340
0 45 3 2
17
```

LAB4

Binary files 1

- Files that store *binary* data are opened using the 'b' flag (instead of 't' for Text):

```
f = open('data.dat', 'br')
```

- For text files, we read and write `str` objects. For binary files, use the `bytes` type instead.
- By default, store data in text files. Text files are human readable (that's good) but take more disk space than binary files.
- Reading and writing binary data is outside the scope of this introductory module. If you read arbitrary binary data, you may need the `struct` module.
- For large/complex scientific data, consider HDF5.

- If you need to store large and/or complex data, consider the use of HDF5 files: <https://portal.hdfgroup.org/display/HDF5/HDF5>
- Python interface: <https://www.h5py.org> (`import h5py`)
- hdf5 files
 - provide a hierarchical structure (like subdirectories and files)
 - can compress data on the fly
 - supported by many tools
 - standard in some areas of science
 - optimised for large volume of data and effective access

HDF5 files: Example data European XFEL

```
[fangohr@max-display001]/gpfs/xfel/exp/XMPL/201750/p700000/raw/r0002% ls -lh
total 756G
-r--r----- 1 xmpldaq  exfel  7.5G Feb 12 18:50 RAW-R0034-AGIPD00-S00000.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 18:51 RAW-R0034-AGIPD00-S00001.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 18:52 RAW-R0034-AGIPD00-S00002.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 19:13 RAW-R0034-AGIPD02-S00003.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 19:14 RAW-R0034-AGIPD02-S00004.h5
-r--r----- 1 xmpldaq  exfel  7.4G Feb 12 19:15 RAW-R0034-AGIPD02-S00005.h5
-r--r----- 1 xmpldaq  exfel  194K Feb 12 19:15 RAW-R0034-AGIPD02-S00006.h5
-r--r----- 1 xmpldaq  exfel  7.5G Feb 12 18:02 RAW-R0034-AGIPD03-S00000.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 18:03 RAW-R0034-AGIPD03-S00001.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 18:04 RAW-R0034-AGIPD03-S00002.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 18:05 RAW-R0034-AGIPD03-S00003.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 18:06 RAW-R0034-AGIPD03-S00004.h5
-r--r----- 1 xmpldaq  exfel  7.5G Feb 12 18:07 RAW-R0034-AGIPD03-S00005.h5
-r--r----- 1 xmpldaq  exfel  194K Feb 12 18:07 RAW-R0034-AGIPD03-S00006.h5
-r--r----- 1 xmpldaq  exfel  7.5G Feb 12 19:28 RAW-R0034-AGIPD04-S00000.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 19:30 RAW-R0034-AGIPD04-S00001.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 19:31 RAW-R0034-AGIPD04-S00002.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 19:32 RAW-R0034-AGIPD04-S00003.h5
-r--r----- 1 xmpldaq  exfel  8.1G Feb 12 19:33 RAW-R0034-AGIPD04-S00004.h5
-r--r----- 1 xmpldaq  exfel  7.5G Feb 12 19:34 RAW-R0034-AGIPD04-S00005.h5
-r--r----- 1 xmpldaq  exfel  194K Feb 12 19:34 RAW-R0034-AGIPD04-S00006.h5
```

HDF5 files: Example data European XFEL

```
[fangohr@max-display001]/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002% lsxfel  
r0002 : Run directory
```

```
# of trains:      3392  
Duration:        0:05:39.100000  
First train ID:  79726751  
Last train ID:   79730142
```

```
16 detector modules (SPB_DET_AGIPD1M-1)  
  e.g. module SPB_DET_AGIPD1M-1 0 : 512 x 128 pixels  
  64 frames per train, 191872 total frames
```

```
3 instrument sources (excluding detectors):  
- SA1_XTD2_XGM/XGM/DOOCS:output  
- SPB_IRU_SIDEMIC_CAM:daqOutput  
- SPB_XTD9_XGM/XGM/DOOCS:output
```

```
13 control sources:  
- ACC_SYS_DOOCS/CTRL/BEAMCONDITIONS  
- SA1_XTD2_XGM/XGM/DOOCS  
- SPB_IRU_AGIPD1M/PSC/HV  
- SPB_IRU_AGIPD1M/TSENS/H1_T_EXTHOUS  
- SPB_IRU_AGIPD1M/TSENS/H2_T_EXTHOUS  
- SPB_IRU_AGIPD1M/TSENS/Q1_T_BLOCK  
- SPB_IRU_AGIPD1M/TSENS/Q2_T_BLOCK  
- SPB_IRU_AGIPD1M/TSENS/Q3_T_BLOCK  
- SPB_IRU_AGIPD1M/TSENS/Q4_T_BLOCK  
- SPB_IRU_AGIPD1M1/CTRL/MC1
```


Exceptions

Exceptions

- Errors arising during the execution of a program result in “exceptions” being ‘raised’ (or ‘thrown’).
- We have seen exceptions before, for example when dividing by zero:

```
>>> 4.5 / 0
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: float division by zero
```

```
or when we try to access an undefined variable:
```

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

- Exceptions are a modern way of dealing with error situations
- We will now see
 - what exceptions are coming with Python
 - how we can “catch” exceptions
 - how we can raise (“throw”) exceptions in our code

In-built Python exceptions

Python's in-built exceptions (from <https://docs.python.org/3/library/exceptions.html>)

BaseException

- +-- SystemExit
- +-- KeyboardInterrupt
- +-- GeneratorExit
- +-- Exception
 - +-- StopIteration
 - +-- StopAsyncIteration
 - +-- ArithmeticError
 - | +-- FloatingPointError
 - | +-- OverflowError
 - | +-- ZeroDivisionError
 - +-- AssertionError
 - +-- AttributeError
 - +-- BufferError
 - +-- EOFError

```
+-- ImportError
|   +-- ModuleNotFoundError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- MemoryError
+-- NameError
|   +-- UnboundLocalError
+-- OSError
|   +-- BlockingIOError
|   +-- ChildProcessError
|   +-- ConnectionError
|       +-- BrokenPipeError
|       +-- ConnectionAbortedError
|       +-- ConnectionRefusedError
|       +-- ConnectionResetError
|   +-- FileExistsError
|   +-- FileNotFoundError
|   +-- InterruptedError
|   +-- IsADirectoryError
|   +-- NotADirectoryError
```

- | +-- PermissionError
- | +-- ProcessLookupError
- | +-- TimeoutError
- +-- ReferenceError
- +-- RuntimeError
 - | +-- NotImplementedError
 - | +-- RecursionError
- +-- SyntaxError
 - | +-- IndentationError
 - | +-- TabError
- +-- SystemError
- +-- TypeError
- +-- ValueError
 - | +-- UnicodeError
 - | +-- UnicodeDecodeError
 - | +-- UnicodeEncodeError
 - | +-- UnicodeTranslateError
- +-- Warning
 - +-- DeprecationWarning
 - +-- PendingDeprecationWarning
 - +-- RuntimeWarning

```
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportError
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning
```

Somewhat advanced use of Python: We can provide our own exception classes (by inheriting from `Exception`).

Exceptions example

- suppose we try to read data from a file:

```
f = open('myfilename.dat', 'r')
for line in f.readlines():
    print(line)
```

- If the file doesn't exist, then the `open()` function raises the `FileNotFoundError` exception:

```
FileNotFoundError: [Errno 2] No such file
↪ or directory: 'myfilename.txt'
```

Catching exceptions

- We can modify our code to 'catch' this error:

```
1  try:
2      f = open('xmyfilename.txt', 'r')
3  except FileNotFoundError:
4      print("The file couldn't be found.")
5  else:
6      # this is executed if no exception is raised
7      for line in f:
8          print(line)
9      f.close()
```

which produces this message:

```
The file couldn't be found.
```

- The `try` branch (line 3) will be executed.

- Should an `FileNotFoundError` exception be raised, then the `except` branch (starting line 5) will be executed.
- Should no exception be raised in the `try` branch, then the `except` branch is ignored, and the program carries on starting in line 9.
- the `sys.exit(n)` function call stops the program, and returns the value of the integer `n` to the operating system as an error code.

Slight extension to print more detailed error message:

```
1  try:
2      f = open('myfilename.txt', 'r')
3  except FileNotFoundError as error:
4      print("The file couldn't be found.")
5      print(f"Error message: {error}")
6  else:
7      # this is executed if no exception is raised
8      for line in f:
9          print(line)
10     f.close()
```

Output:

The file couldn't be found.

Error message: [Errno 2] No such file or directory:

↪ 'myfilename.txt'

Catching exceptions summary

- Catching exceptions allows us to take action on errors that occur
 - For the file-reading example, we could ask the user to provide another file name if the file can't be opened.
- Catching an exception once an error has occurred may be easier than checking beforehand whether a problem will occur (*"It is easier to ask forgiveness than get permission".*)

Overview try-except-else-finally

```
try:
    # statement that might raise an exception
    pass
except SomeError:
    # deal with error
    pass
else:
    # code to execute if no error is raised
    pass
finally:
    # code that must always be executed
    # (for example closing a file)
    pass
```

try-except example

From [Python documentation](#)

```
try:
    f = open("myfile.txt")
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

The last `raise` re-raises the last exception as if it wasn't caught before.

Raising exceptions

- Because exceptions are Python's way of dealing with runtime errors, we should use exceptions to report errors that occur in our own code.
- To raise a `ValueError` exception, we use

```
raise ValueError("Message")
```

and can attach a message "Message" (of type string) to that exception which can be seen when the exception is reported or caught:

```
>>> raise ValueError("Some problem occurred")
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: Some problem occurred
```


Raising NotImplementedError Example

Often used is the `NotImplementedError` in *incremental software development*:

```
def my_complicated_function(x):  
    message = f"Called with x={x}"  
    raise NotImplementedError(message)
```

If we call the function:

```
>>> my_complicated_function(42)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 3, in my_complicated_function  
NotImplementedError: Called with x=42
```

Extend `print_line_sum_of_file(filename)` so that if the data file contains non-numbers (i.e. strings), these evaluate to the value 0. For example

1 2 4 -> 7

1 cat 4 -> 5

coffee -> 0

LAB5

Printing

- the `print` function sends content to the “standard output” (usually the screen)
- `print()` prints an empty line:

```
>>> print()
```

- Given a single string argument, this is printed, followed by a new line character:

```
>>> print("Hello")
```

```
Hello
```

- Given another object (not a string), the `print` function will *ask* the object for its preferred way to be represented as a string (via the `__str__` method):

```
>>> print(42)
```

```
42
```

- Given multiple objects separated by commas, they will be printed separated by a space character:

```
>>> print("dog", "cat", 42)
```

```
dog cat 42
```

- To suppress printing of a new line, use the `end=''` option:

```
>>> print("Dog", end=''); print("Cat")
```

```
DogCat
```

```
>>>
```

- Or customise the token added at the end of the string.

```
>>> print("Dog", end=' [bark]\n')
Dog [bark]
```

- We can also redirect printing into file objects:

```
>>> with open("test.txt", "tw") as f:
...     print("Hello World", file=f)
...
>>> with open("test.txt") as f:
...     f.read()
...
'Hello World\n'
```

Common strategy for the print command

- Construct some string `s`, then print this string using the `print` function

```
>>> s = "I am the string to be printed"
```

```
>>> print(s)
```

```
I am the string to be printed
```

- The question is, how can we construct the string `s`? We talk about string formatting.

- 1. 1991: % operator (Python 1 and 2)
- 2. 2006: `str.format()` “new style” or “advanced” string formatting (Python 3)
- 3. 2016: f-strings (Python 3.6)

1. String formatting: the percentage (%) operator

% operator syntax

Syntax: `A % B`

where **A** is a string, and **B** a Python object, or a tuple of Python objects.

The format string **A** needs to contain k format specifiers if the tuple has length k . The operation returns a string.

Example: basic formatting of one number

```
>>> import math
>>> p = math.pi
>>> "%f" % p          # format p as float (%f)
'3.141593'           # returns string
>>> "%d" % p          # format p as integer (%d)
'3'
>>> "%e" % p          # format p in exponential style
'3.141593e+00'
>>> "%g" % p          # format using fewer characters
'3.14159'
```

The format specifiers can be combined with arbitrary characters in string:

```
>>> 'the value of pi is approx %f' % p
'the value of pi is approx 3.141593'
>>> '%d is my preferred number' % 42
'42 is my preferred number'
```

Combining multiple objects

```
>>> "%d times %d is %d" % (10, 42, 10 * 42)
'10 times 42 is 420'
>>> "pi=%f and 3*pi=%f is approx 10" % (p, 3*p)
'pi=3.141593 and 3*pi=9.424778 is approx 10'
```

Fixing width and/or precision of resulting string

```
>>> '%f' % 3.14      # default width and precision  
'3.140000'
```

```
>>> '%10f' % 3.14   # 10 characters long  
' 3.140000'
```

```
>>> '%10.2f' % 3.14 # 10 long, 2 post-dec digits  
'      3.14'
```

```
>>> '%.2f' % 3.14   # 2 post-decimal digits  
'3.14'
```

```
>>> '%.14f' % 3.14 # 14 post-decimal digits
'3.1400000000000000'
```

There is also the format specifier %s that expects a string, or an object that can provide its own string representation.

Combined with a width specifier, this can be used to align columns of strings in tables:

```
>>> "%10s" % "apple"
'   apple'
>>> "%10s" % "banana"
'   banana'
```

Common formatting specifiers

A list of common formatting specifiers, with example output for the astronomical unit (AU) which is the distance from Earth to Sun [in metres]:

```
>>> AU = 149597870700 # astronomical unit [m]
>>> "%f" % AU         # line 1 in table
'149597870700.000000'
```

specifier	style	Example output for AU
%f	floating point	149597870700.000000
%e	exponential notation	1.495979e+11
%g	shorter of %e or %f	1.49598e+11
%d	integer	149597870700
%s	str()	149597870700
%r	repr()	149597870700

Summary %-operator for printing

Create string using the %-operator, then pass the string to the print function. Typically done in the same line:

```
>>> import math
>>> print("My pi = %.2f." % math.pi)
My pi = 3.14.
```

Print multiple values:

```
>>> print("a=%d b=%d" % (10, 20))
a=10 b=20
```

Very similar syntax exists in other languages, for example C and Matlab, for formatted data output to screen and files.

2. New style string formatting (format method)

A new(er) system of built-in formatting has been proposed (PEP3101), titled **Advanced String Formatting** and is available in Python 3.

Basic ideas in examples:

- Pairs of curly braces are the placeholders.

```
>>> "{} owns {} bikes".format('Peter', 4)
'Peter owns 4 bikes'
```

- Formatting behaviour of %f can be achieved through `{:f}`, (same for %d, %e, etc)

```
>>> "Pi is approx {:f}.".format(math.pi)
'Pi is approx 3.141593.'
```

- Width and post decimal digits can be specified as before:
>>> "Pi is approx {:.2f}.".format(math.pi)
'Pi is approx 3.14.'
>>> "Pi is approx {:.2f}.".format(math.pi)
'Pi is approx 3.14.'

Further Reading

- Examples
<http://docs.python.org/library/string.html#format-examples>
- [Python Enhancement Proposal 3101](#)

3. f-strings (formatted string literals)

- Introduced in Python 3.6
- Described in PEP498
<https://www.python.org/dev/peps/pep-0498/>
- combines with `str.format` syntax

f-strings examples

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> value = 12.34567
>>> f"result: {value}"
'result: 12.34567'
```

f-strings re-use new style syntax

We can combine f-strings with new format specifiers:

```
>>> value = 12.34567
>>> f"result: {value:10}" # 10 spaces
'result: 12.34567'
>>> f"result: {value:e}" # %e behaviour
'result: 1.234567e+01'
>>> f"result: {value:f}" # %f behaviour
'result: 12.345670'
>>> f"result: {value:.4f}" # 4 post-decimal digits
'result: 12.3457'
>>> f"result: {value:.4}" # 4 digits precision
'result: 12.35'
```

Expressions in f-strings are evaluated at run time

We can evaluate Python expressions in the f-strings:

```
>>> import math
>>> f"The diagonal has length {math.sqrt(2)}."
'The diagonal has length 1.4142135623730951.'
```

(Advanced:) Precision specifier can be variables:

```
>>> width = 10
>>> precision = 4
>>> f"{math.pi:{width}}.{precision}"
'      3.142'
```

Show variable name and value with {name=}

Convenient short cut for debugging print statements:

```
>>> a = 10
>>> b = 20
>>> c = math.sqrt(a**2 + b**2)
>>> f"State: {a=} {b=} {c=}"
'State: a=10 b=20 c=22.360679774997898'
```


Comparison string formatting generations 1

Example 1

```
>>> value = 42
```

```
>>> "the value is %s" % value  
'the value is 42'
```

```
>>> "the value is {}".format(value)  
'the value is 42'
```

```
>>> f"the value is {value}"  
'the value is 42'
```

Comparison string formatting generations 2

Example 2

```
>>> import math
```

```
>>> x = math.pi
```

```
# conventional:
```

```
>>> "x is %f and x^2 is approx %.1f" % (x, x**2)
```

```
'x is 3.141593 and x^2 is approx 9.9'
```

```
# new-style:
```

```
>>> "x is {:f} and x^2 is approx {:.1f}".format(x, x**2)
```

```
'x is 3.141593 and x^2 is approx 9.9'
```

```
# f-strings:
```

```
>>> f"x is {x:f} and x^2 is approx {x**2:.1f}"
```

```
'x is 3.141593 and x^2 is approx 9.9'
```

```
>>> f"{x=:f} and {x**2=:.1f}" # alternative simplification
```

```
'x=3.141593 and x**2=9.9'
```

What formatting should I use?

- use f-strings if you can
- The `.format` method more elegant and versatile than `%`
- `%` operator style okay, links to Matlab, C, ...
- Choice partly a matter of taste, history and existing code
 - do your collaborators know the method you use?
 - Should be aware (in a passive sense) of different possible styles (so we can read code from others)

Changes from Python 2 to Python 3: print

One (maybe the most obvious) change going from Python 2 to Python 3 is that the `print` command loses its special status. In Python 2, we could print "Hello World" using

```
print "Hello World"           # allowed in Python 2
```

Effectively, we call the function `print` with the argument "Hello World". All other functions in Python are called such that the argument is enclosed in parentheses, i.e.

```
print("Hello World")         # required in Python 3
```

This is the new convention *required* in Python 3 (and *allowed* for recent version of Python 2.x.)

The `str` function and `__str__` method

All objects in Python should provide a method `__str__` which returns an *informal* string representation of the object.

This method `a.__str__` is called when we apply the `str` function to object `a`:

```
>>> a = 3.14
>>> a.__str__()
'3.14'
>>> str(a)
'3.14'

>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2022, 1, 13, 13, 44, 56, 392268)
>>> str(now)
'2022-01-13 13:44:56.392268'
```

Implicit calling of `str` function

The string method `x.__str__` of object `x` is called implicitly, when we

- use the `"%s"` format specifier in %-operator formatting to print `x`
- use the `"{}"` format specifier in `.format` to print `x`
- use the `"{x}"` notation in f-strings
- pass the object `x` directly to the `print` command

```
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2022, 1, 13, 13, 44, 56, 392268)
>>> "%s" % now
'2022-01-13 13:44:56.392268'
>>> "{}".format(now)
'2022-01-13 13:44:56.392268'
>>> f"{now}"
'2022-01-13 13:44:56.392268'
>>> print(now)
2022-01-13 13:44:56.392268
```

The `repr` function and `__repr__` method

- The `repr` function should convert a given object into an *as accurate as possible* string representation
- The `repr` function will generally provide a more detailed string than `str`.
- Applying `repr` to the object `x` will attempt to call `x.__repr__()`.
- The python (and IPython) prompt uses `repr` to 'display' objects.

Example:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> str(t)
'2022-01-13 13:55:39.158456'
>>> repr(t)
'datetime.datetime(2022, 1, 13, 13, 55, 39, 158456)'
>>> t
datetime.datetime(2022, 1, 13, 13, 55, 39, 158456)
```

For many objects, `str(x)` and `repr(x)` return the same string.

The eval function

The `eval` function accepts a string, and *evaluates* the string (as if it was entered at the Python prompt):

```
>>> x = 1
>>> eval('x + 1')
2
>>> s = "[10, 20, 30]"
>>> type(s)
<class str>
>>> eval(s)
[10, 20, 30]
>>> type(eval(s))
<class list>
```

The repr and eval function

Given an accurate representation of an object as a string, we can convert that string into the object using the `eval` function.

```
>>> i = 42
>>> type(i)
<class int>
>>> repr(i)
'42'
>>> type(repr(i))
<class str>
>>> eval(repr(i))
42
>>> type(eval(repr(i)))
<class int>
```

The datetime example:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> t_as_string = repr(t)
>>> t_as_string
'datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)'
>>> t2 = eval(t_as_string)
>>> t2
datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)
>>> type(t2)
<class datetime.datetime>
>>> t == t2
True
```

Higher Order Functions

Motivational exercise: function tables

- Write a function `print_x2_table()` that prints a table of values of $f(x) = x^2$ for $x = 0, 0.5, 1.0, \dots, 2.5$, i.e.

```
0.0 0.0
0.5 0.25
1.0 1.0
1.5 2.25
2.0 4.0
2.5 6.25
```

- Then do the same for $f(x) = x^3$
- Then do the same for $f(x) = \sin(x)$

Can we avoid code duplication?

Idea: Pass function $f(x)$ to tabulate to tabulating function

Example: (`print_f_table.py`)

```
def print_f_table(f):  
    for i in range(6):  
        x = i * 0.5  
        fx = f(x)  
        print(f"{x} {fx}")
```

```
def square(x):  
    return x ** 2
```

```
print_f_table(square)
```

produces

```
0.0 0.0
```

```
0.5 0.25
```

```
1.0 1.0
```

```
1.5 2.25
```

```
2.0 4.0
```

```
2.5 6.25
```

Can we avoid code duplication (2)?

```
def print_f_table(f):  
    for i in range(6):  
        x = i * 0.5  
        fx = f(x)  
        print(f"{x} {fx}")
```

```
def square(x):  
    return x ** 2
```

```
def cubic(x):  
    return x ** 3
```



```
print("Square"); print_f_table(square)
print("Cubic");  print_f_table(cubic)
```

produces:

Square

0.0 0.0

0.5 0.25

1.0 1.0

1.5 2.25

2.0 4.0

2.5 6.25

Cubic

0.0 0.0

0.5 0.125

1.0 1.0

1.5 3.375

2.0 8.0

2.5 15.625

Example: iterating over functions

- Example (trigtable.py):

```
import math
funcs = [math.sin, math.cos]
for f in funcs:
    fname = f.__name__
    for x in [0, math.pi/2]:
        fx = f(x)
        print(f"{fname}({x:.3f}) = {fx:.3f}")
```

produces

```
sin(0.000) = 0.000
sin(1.571) = 1.000
cos(0.000) = 1.000
cos(1.571) = 0.000
```

Higher order functions / are first class objects

Functions are 'just' objects in Python. Related terminology:

- Functions are *first class objects* \leftrightarrow functions can be given to other functions as arguments
- *Higher order functions* accept (or return) functions as arguments.

Modules

- Motivation: it is useful to bundle functions that are used repeatedly and belong to the same subject area into one module file (also called “library”)
- This allows to re-use the functions in multiple Python applications.
- Every Python file can be imported as a module.
- If the module file contains commands (other than class and function *definitions*) then these are executed when the file is imported. This can be desired but sometimes it is not.

The internal `__name__` variable (1)

- Here is an example of a module file saved as `module1.py`:

```
def someusefultion():  
    pass
```

```
print(f"My name is {__name__}")
```

We can execute this module file, and the output is

```
My name is __main__
```

- The internal variable `__name__` takes the (string) value `"__main__"` if the program file `module1.py` is executed.

- On the other hand, we can *import module1.py* in another file, for example like this:

```
import module1
```

The output is now:

```
My name is module1
```

- We see that `__name__` inside a module takes the value of the module name if the file is imported.

```
if __name__ == "__main__" ...
```

module2.py:

```
1 def someusefulfunction():
2     pass
3
4 if __name__ == "__main__":
5     print("I am the top level")
6 else:
7     print(f"I am imported as a library '{__name__}')
```

- Line 5 is only executed when the module is executed as the top level (for example as `python module2.py`, or pressing F5 in Spyder when editing the file `module2.py`).
- `__name__` allows conditional execution of code when top-level or imported.

Application file example

```
def useful_function():
    # Core function in this app.
    # Could be useful in other apps.
    pass

def main():
    # Main functionality of this app in here.
    x = useful_function()
    # ...

if __name__ == "__main__":
    main() # start main application
else:
    # get here if the file is imported
    pass
```

Library file example

```
def useful_function():
    # core functionality of library here
    pass

def test_for_useful_function():
    print("Running self test ...")

if __name__ == "__main__":
    test_for_useful_function()
else:
    print("Setting up library")
    # initialisation code that might be needed
    # if imported as a library
```

Default arguments

Default argument values

- Motivation:
 - suppose we need to compute the area of rectangles and
 - we know the side lengths **a** and **b**.
 - Most of the time, **b=1** but sometimes **b** can take other values.
- Solution 1:

```
def area(a, b):  
    return a * b
```

```
print(f"The area is {area(3, 1)}")  
print(f"The area is {area(2.5, 1)}")  
print(f"The area is {area(2.5, 2)}")
```

- We can make the function more user friendly by providing a *default* value for **b**. We then only have to specify **b** if it is different from this default value:
- Solution 2 (with default value for argument **b**):

```
def area(a, b=1):  
    return a * b
```

```
print(f"The area is {area(3, 1)}")  
print(f"The area is {area(2.5, 1)}")  
print(f"The area is {area(2.5, 2)}")
```

- If a default value is defined, then this parameter (here **b**) is optional when the function is called.
- Default parameters have to be at the end of the argument list in the function definition.

You may have met default arguments in use before, for example

- the `print` function uses `end='\n'` as a default value
- the `list.pop` method uses `index=-1` as a default

LAB6

Keyword argument values

- We can call functions with a “keyword” and a value. (The keyword is the name of the variable in the function definition.)
- Here is an example

```
def f(a, b, c):  
    print(f"{a=} {b=} {c=}")
```

```
f(1, 2, 3)
```

```
f(c=3, a=1, b=2)
```

```
f(1, c=3, b=2)
```

which produces this output:

a=1 b=2 c=3

a=1 b=2 c=3

a=1 b=2 c=3

- If we use *only* keyword arguments in the function call, then we do not need to know the *order* of the arguments. (This is good.)
- Choosing meaningful variable names in the function definition makes the function more user friendly.

Combining keyword arguments with default argument values

- Can combine default value arguments and keyword arguments
- Example: Imagine for a numerical integration routine we use 100 subdivisions unless the user provides a number

```
def trapez(function, a, b, subdivisions=100):  
    # code missing here  
    pass
```

```
import math  
int1 = trapez(a=0, b=10, function=math.sin)  
int2 = trapez(b=0, function=math.exp,  
              subdivisions=1000, a=-0.5)
```

Advanced: disallow or enforce keyword argument use

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           | Positional or keyword |
    |           |           |
    |           |           | - Keyword only
    -- Positional only
```

See [https:](https://www.python.org/dev/peps/pep-0570/#how-to-teach-this)

[//www.python.org/dev/peps/pep-0570/#how-to-teach-this](https://www.python.org/dev/peps/pep-0570/#how-to-teach-this)

```
def standard_arg(arg):  
    print(arg)  
  
def pos_only_arg(arg, /):  
    print(arg)  
  
def kwd_only_arg(*, arg):  
    print(arg)  
  
def combined_example(pos_only, /, standard, *, kwd_only):  
    print(pos_only, standard, kwd_only)
```


Namespaces

We distinguish between

- *global* variables (defined in main program) and
- *local* variables (defined for example in functions)
- *built-in* functions

Python's look up rule for Names

When coming across an identifier, Python looks for this in the following order in

- the local name space (L)
- (if appropriate in the next higher level local name space),
(L², L³, ...)
- the global name space (G)
- the set of built-in commands (B)

This is summarised as “LGB” or “LⁿGB”.

If the identifier cannot be found, a `NameError` is raised.

Local names shadow global names

- This means, we can *read* global variables from functions.
Example:

```
def f():  
    print(x)
```

```
x = 'I am global'  
f()
```

Output:

```
I am global
```


- but local variables “shadow” global variables:

```
def f():  
    y = 'I am local y'  
    print(x)  
    print(y)
```

```
x = 'I am global x'  
y = 'I am global y'  
f()  
print("back in main:")  
print(y)
```

Output:

```
I am global x  
I am local y  
back in main:  
I am global y
```

- To *modify* global variables within a local namespace, we need to use the `global` keyword.

(This is not recommended so we won't explain it. See also next slide.)

Why should I care about global variables?

- Generally, the use of global variables is not recommended:
 - functions should take all necessary input as arguments and
 - return all relevant output.
 - This makes the functions work as independent units which is good engineering practice and essential to control complexity of software.
- However, sometimes the same constant or variable (such as the mass of an object) is required throughout a program:

- it is not good practice to define this variable more than once (it is likely that we assign different values and get inconsistent results)
- in this case — in small programs — the use of (read-only) global variables may be acceptable.
- Object Oriented Programming provides a somewhat neater solution to this.

Editors and IDEs

IPython (interactive python)

- Interactive Python (`ipython` from Command Prompt/Unix-shell)
- command history (across sessions), auto completion,
- special commands:
 - `%run myfile` will execute file `myfile.py` in current name space
 - `%reset` can delete all objects if required
 - use `range?` instead of `help(range)`
 - `%logstart` will log your session
 - `%prun` will profile code
 - `%timeit` can measure execution time
 - `%load` loads file for editing (also from URL)
 - `%debug` start debugger after error
 - `%pdb` automatic calling of debugger
- Much more (read at <http://ipython.org>)

Jupyter Notebook useful for research and data science

- Used to be the IPython Notebook, but now supports many more languages (JULia, PYThon, ER → JUPYTER)
- Notebook is *executable* document hosted in web browser.
- Home page <http://jupyter.org>

Great value for research

- Fangohr et al: *Data Exploration and Analysis with Jupyter Notebooks* [10.18429/JACoW-ICALEPCS2019-TUCPR02](https://doi.org/10.18429/JACoW-ICALEPCS2019-TUCPR02) (2020)
- Granger and Perez: *Thinking and Storytelling with Jupyter*, [10.1109/MCSE.2021.3059263](https://doi.org/10.1109/MCSE.2021.3059263) (2021)
- Fangohr, Di Pierro and Kluyver: *Jupyter in Computational Science*, [10.1109/MCSE.2021.3059494](https://doi.org/10.1109/MCSE.2021.3059494) (2021)
- Beg, Fangohr, et al: *Using Jupyter for reproducible scientific workflows*, Computing in Science and Engineering 23, 36-46 [10.1109/MCSE.2021.3052101](https://doi.org/10.1109/MCSE.2021.3052101) (2021)
- [Blog entry: Jupyter for Computational Science and Data Science](#) (2022)

Including

- Emacs
- vim (vi)
- vim *and* Emacs → Spacemacs
- Spyder
- PyCharm (commercial)
- Visual studio code
- Sublime Text (commercial)
- ...

List comprehension

List comprehension

- List comprehension follows the mathematical “set builder notation”
- Convenient way to process a list into another list (without for-loop).

Examples

```
>>> [2*i for i in range(5)]  
[0, 2, 4, 6, 8]
```

```
>>> [x**2 for x in range(10)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension structure

Structure of list comprehension:

```
[EXPRESSION(OBJECT) for OBJECT in SEQUENCE]
```

where EXPRESSION, OBJECT, and SEQUENCE can vary.

Examples:

```
>>> [2*i for i in range(5)]  
[0, 2, 4, 6, 8]
```

```
>>> import math  
>>> [math.sqrt(x) for x in [1, 4, 9, 16]]  
[1.0, 2.0, 3.0, 4.0]
```

```
>>> [s.capitalize() for s in ["dog", "cat", "mouse"]]  
['Dog', 'Cat', 'Mouse']
```

List comprehension example 1 and 2

Can be useful to populate lists with numbers quickly

- Example 1:

```
>>> ys = [x**2 for x in range(10)]
>>> ys
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Example 2:

```
>>> import math
>>> xs = [0.1 * i for i in range(5)]
>>> xs
[0.0, 0.1, 0.2, 0.3, 0.4]
>>> ys = [math.exp(x) for x in xs]
>>> ys
[1.0, 1.1051709180756477, 1.2214027581601699,
 1.3498588075760032, 1.4918246976412703]
```

List comprehension with filter

```
[EXPRESSION(OBJECT) for OBJECT in SEQUENCE  
if CONDITION(OBJECT)]
```

- include OBJECT only if CONDITION(OBJECT) is True.
- Example:

```
>>> [i for i in range(10)]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> [i for i in range(10) if i > 5]  
[6, 7, 8, 9]
```

```
>>> [i for i in range(10) if i**2 > 5]  
[3, 4, 5, 6, 7, 8, 9]
```

In addition to *list comprehension* there is also *dictionary comprehension* available:

```
>>> {x: x**2 for x in range(5)}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
>>> {word: len(word) for word in ["dog", "bird", "mouse"]}  
{'dog': 3, 'bird': 4, 'mouse': 5}
```

Generator comprehension (advanced)

Generators (see slide 92) can also be created using a comprehension syntax:

```
>>> gen = (x**2 for x in range(5))
>>> type(gen)
<class 'generator'>
>>> for item in gen:
...     print(item)
...
0
1
4
9
16
>>> list( (x**2 for x in range(5)) )
[0, 1, 4, 9, 16]
>>>
```

Dictionaries

Dictionaries

- Python provides another data type: the dictionary.

Dictionaries are also called “associative arrays” and “hash tables”.

- Dictionaries are *unordered* sets of *key-value pairs*.

Starting from Python 3.7, dictionaries preserve insertion order.

- An empty dictionary can be created using curly braces:

```
>>> d = {}
```

- Keyword-value pairs can be added like this:

```
>>> d['today'] = '22 deg C' # 'today' is key
                             # '22 deg C' is value
```

```
>>> d['yesterday'] = '19 deg C'
```

- We can retrieve values by using the keyword as the index:

```
>>> d['today']
'22 deg C'
```

- `d.keys()` returns all keys:

```
>>> d.keys()
['today', 'yesterday']
```

- `d.values()` returns all values:

```
>>> d.values()
['22 deg C', '19 deg C']
```

- Check if key is in dictionary:

```
>>> 'today' in d.keys()
True
```

Equivalent to

```
>>> 'today' in d
True
```

Dictionary example 1

```
order = {} # create empty dictionary

# add orders as they come in
order['Peter'] = 'Sparkling water'
order['Paul'] = 'Half pint of beer'
order['Mary'] = 'Gin tonic'

# deliver order at bar
for person in order.keys():
    print(f"{person} requests {order[person]}")
```

which produces this output:

```
Peter requests sparkling water
Paul requests Half pint of beer
Mary requests Gin tonic
```

Some more technicalities:

- The dictionary key can be any *immutable* Python object. This includes:
 - numbers
 - strings
 - tuples.
- dictionaries are very fast in retrieving values (when given the key)

Dictionary use case

```
# keys are names of people
# values are the room numbers
room = {} # better name: room by person
room["Andy"] = 1031
room["Barbara"] = 1027
room["Charles"] = 1033

for person in room.keys():
    print(f"{person} works in {room[person]}")
```

Output:

```
Andy works in 1031
Barbara works in 1027
Charles works in 1033
```

Without dictionary:

```
people = ["Andy", "Barbara", "Charles"]
rooms = [1031, 1027, 1033]
# possible inconsistency here since we have two lists
if not len(people) == len(rooms):
    raise ValueError("people and rooms differ in length")

for i in range(len(rooms)):
    print(f"{people[i]} works in room {rooms[i]}")
```

Output:

```
Andy works in room 1031
Barbara works in room 1027
Charles works in room 1033
```

Iterating over dictionaries

Iterating over the dictionary itself is equivalent to iterating over the keys. Example:

```
order = {}          # create empty dictionary

# add orders as they come in
order['Peter'] = 'Sparkling water'
order['Paul'] = 'Half pint of beer'
order['Mary'] = 'Gin tonic'

# iterating over keys:
for person in order.keys():
    print(f"{person} requests {order[person]}")

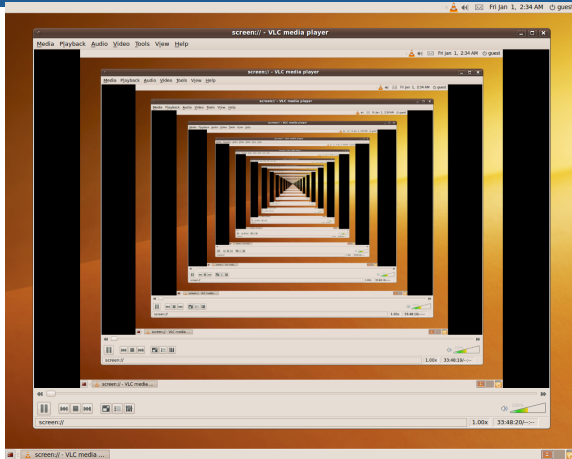
# is equivalent to iterating over the dictionary:
for person in order:
    print(f"{person} requests {order[person]}")
```

What to remember:

- Python provides dictionaries
- very powerful construct
- a bit like a data base (and values can be dictionary objects)
- fast to retrieve value
- likely to be useful if you are dealing with two lists at the same time (possibly one of them contains the keyword and the other the value)
- useful if you have a data set that needs to be indexed by strings or tuples (or other immutable objects)

Recursion

Recursion



Recursion in a screen recording program, where the smaller window contains a snapshot of the entire screen. Source:

<http://en.wikipedia.org/wiki/Recursion>

Recursion example: factorial

- Computing the factorial (i.e. $n!$) can be done by computing $(n - 1)!n$, i.e. we reduce the problem of size n to a problem of size $n - 1$.
- For recursive problems, we always need a *base case*. For the factorial we know that $0! = 1$.
- For $n = 4$:

$$4! = 3! \cdot 4 \tag{1}$$

$$= 2! \cdot 3 \cdot 4 \tag{2}$$

$$= 1! \cdot 2 \cdot 3 \cdot 4 \tag{3}$$

$$= 0! \cdot 1 \cdot 2 \cdot 3 \cdot 4 \tag{4}$$

$$= 1 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \tag{5}$$

$$= 24. \tag{6}$$

Recursion example

Python code to compute the factorial recursively:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

Usage output:

```
>>> factorial(0)  
factorial(0)  
1  
>>> factorial(2)  
2  
>>> factorial(4)  
24
```

Recursion example Fibonacci numbers

Defined (recursively) as $f(n) = f(n - 1) + f(n - 2)$ for integers n , and $n > 0$, and $f(1) = 0$ and $f(2) = 1$

Python implementation (fibonacci.py):

```
def f(n):  
    if n == 1:  
        return 0  
    elif n == 2:  
        return 1  
    else:  
        return f(n - 2) + f(n - 1)
```

1. Write a function `recsum(n)` that sums the numbers from 1 to n *recursively*
2. Study the recursive Fibonacci function from slide 216:
 - what is the largest number n for which we can reasonable compute $f(n)$ within a minute?
 - Can you write faster versions of the Fibonacci function? (There are faster versions with and without recursion.)

Common Computational Tasks

Overview common computational tasks

- Data file processing, python & numpy (array)
- Random number generation and Fourier transforms (numpy)
- Linear algebra (numpy)
- Interpolation of data (`scipy.interpolate.interp`)
- Fitting a curve to data (`scipy.optimize.curve_fit`)
- Integrating a function numerically (`scipy.integrate.quad`)
- Integrating a ordinary differential equation numerically (`scipy.integrate.solve_ivp`)

- Finding the root of a function
(`scipy.optimize.fsolve`,
`scipy.optimize.brentq`)
- Minimising a function (`scipy.optimize.fmin`)
- Symbolic manipulation of terms, including integration,
differentiation and code generation (`sympy`)
- Data science, processing, cleaning and analysing data,
tabular data (`pandas`)

Root finding

Root finding

Given a function $f(x)$, we are searching an x_0 so $f(x_0) = 0$. We call x_0 a root of $f(x)$.

Why?

- Often need to know when a particular function reaches a value, for example the water temperature $T(t)$ reaching 373 K. In that case, we define

$$f(t) = T(t) - 373$$

and search the root t_0 for $f(t)$

We introduce two methods:

- Bisection method
- Newton method

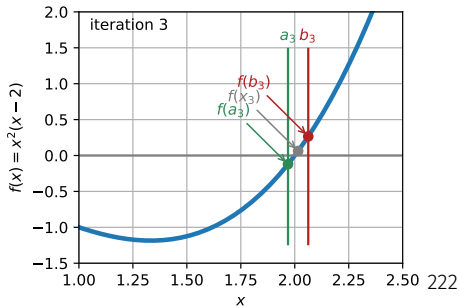
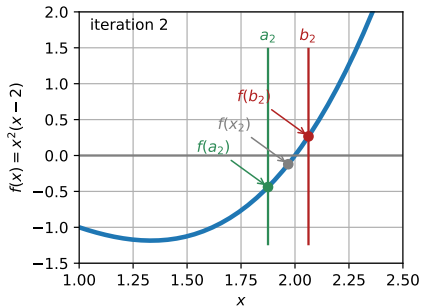
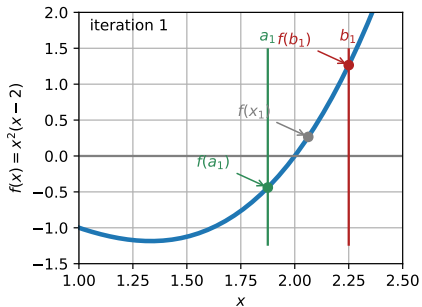
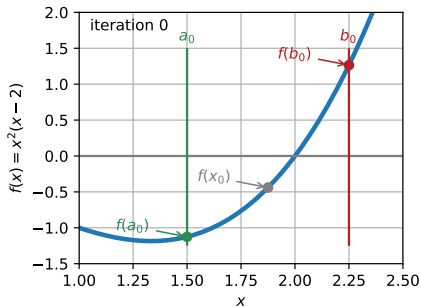
The bisection algorithm

- Function: `bisect(f, a, b)`
- Assumptions:
 - Given: a (float)
 - Given: b (float)
 - Given: $f(x)$, continuous with single root in $[a, b]$, i.e. $f(a)f(b) < 0$
 - Given: f_{tol} (float), for example $f_{\text{tol}} = 10^{-6}$

The bisection method returns x so that $|f(x)| < f_{\text{tol}}$.

1. $x = (a + b)/2$
2. while $|f(x)| > f_{\text{tol}}$ do
 - if $f(x)f(a) > 0$
then $a \leftarrow x$ # throw away left half
 - else $b \leftarrow x$ # throw away right half
 - $x = (a + b)/2$
3. return x

The bisection algorithm

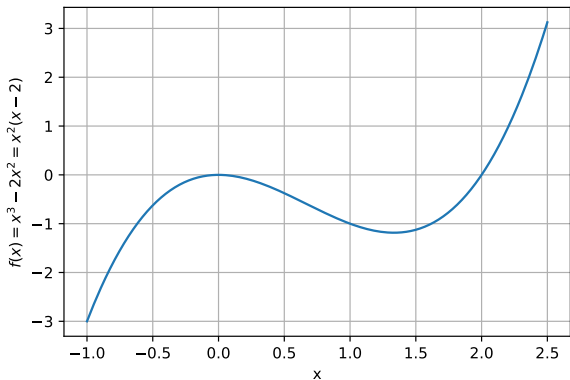


The bisection function from `scipy`

- Scientific Python provides an interface to the “Minpack” library. One of the functions is
- `scipy.optimize.bisect(f, a, b[, xtol])`
 - `f` is the function for which we search x such that $f(x) = 0$
 - `a` is the lower limit of the bracket $[a, b]$ around the root
 - `b` is the upper limit of the bracket $[a, b]$ around the root
 - `xtol` is an *optional* parameter that can be used to modify the default accuracy of $x_{tol} = 10^{-12}$
- the `bisect` function stops ‘bisecting’ the interval around the root when $|b-a| < xtol$.

Example

- Find root of function $f(x) = x^2(x - 2)$
- f has a double root at $x = 0$, and a single root at $x = 2$.
- Ask algorithm to find single root at $x = 2$.



Using bisection algorithm from scipy

```
from scipy.optimize import bisect

def f(x):
    """returns f(x)=x^3-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x ** 3 - 2 * x ** 2

# main program starts here
x = bisect(f, a=1.5, b=3, xtol=1e-6)

print(f"Root is approx {x}.")
print(f"The exact error is {x-2}.")
print(f"Error is less than 1e-6: {abs(x-2)<1e-6}")
```

produces

```
Root is approx 2.000000238418579.
The exact error is 2.384185791015625e-07.
Error is less than 1e-6: True
```

The Newton method

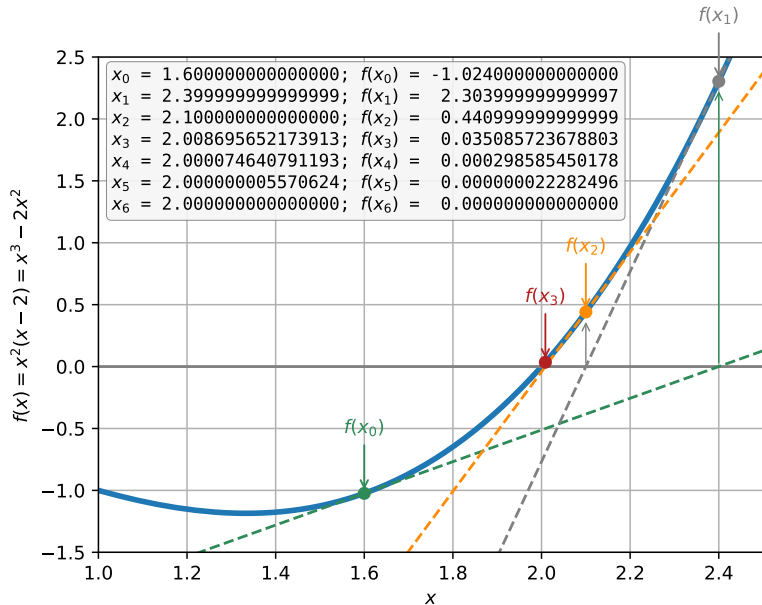
- Newton method for root finding: find x_0 so that $f(x_0) = 0$.
- Idea: close to the root, the tangent of $f(x)$ is likely to point to the root. Make use of this information.
- Algorithm:
while $|f(x)| > \text{ftol}$, do

$$x \leftarrow x - \frac{f(x)}{f'(x)}$$

where $f'(x) = \frac{df}{dx}(x)$.

- Much better convergence than bisection method
- but not guaranteed to converge.
- Need a good initial guess x for the root.
- Need a way to compute (approximate) $f' \equiv \frac{df}{dx}(x)$.

The Newton method (tol=1e-15)



Using Newton algorithm from scipy

```
from scipy.optimize import newton

def f(x):
    """returns f(x)=x^3-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x ** 3 - 2 * x ** 2

# main program starts here
x = newton(f, x0=1.6, tol=1e-6)

print(f"Root is approx {x}.")
print(f"The exact error is {2-x}.")
```

produces

```
Root is approx 1.9999999999999902.
The exact error is 9.769962616701378e-15.
```

Uses [Secant method](#) as f' is not known (and not passed to newton).

Bisection method

- Requires root in bracket $[a, b]$
- guaranteed to converge (for single roots)
- Library function:
`scipy.optimize.bisect`

Newton method

- Requires good initial guess x for root x_0
- may never converge
- but if it does, it is quicker than the bisection method
- Library function:
`scipy.optimize.newton`

Root finding summary

- Given the function $f(x)$, applications for root finding include:
 - to find x_1 so that $f(x_1) = y$ for a given y (this is equivalent to computing the inverse of the function f).
 - to find crossing point x_c of two functions $f_1(x)$ and $f_2(x)$ (by finding root of difference function $g(x) = f_1(x) - f_2(x)$)
- Recommended method: `scipy.optimize.brentq` which combines the safe feature of the bisection method with the speed of the Newton method.
- For multi-dimensional functions $f(\mathbf{x})$, use `scipy.optimize.fsolve`.

Using BrentQ algorithm from scipy

```
from scipy.optimize import brentq

def f(x):
    """returns f(x)=x^3-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x ** 3 - 2 * x ** 2

# main program starts here
x = brentq(f, a=1.5, b=3, xtol=1e-6)

print(f"Root is approx {x}.")
print(f"The exact error is {2-x}.")
```

produces:

```
Root is approx 2.0000000189582865.
The exact error is -1.8958286496228993e-08.
```


Using fsolve algorithm from scipy

```
from scipy.optimize import fsolve # multidimensional solver

def f(x):
    """returns f(x)=x^2-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x**3 - 2 * x**2

sol = fsolve(f, x0=[1.6])
x = sol[0] # root x is first entry
print(f"Returned object is {sol}.")
print(f"Root is approx {x}")
print(f"The exact error is {2-x}.")
```

produces:

```
Returned object is sol=array([2.]).
Root is approx. 2.0
The exact error is 0.0.
```

Example `fsolve` for multi-dimensional optimisation problem

```
from scipy.optimize import fsolve # multidimensional solver

def f(v):
    """Return f(x, y) = (x^3, y). Trivial example with
    root at x=0 and y=-1"""
    x, y = v
    return x**3, y+1

x, y = fsolve(f, x0=[2, 2]) # start search from x=2, y=2
print(f"Root is approx. {x=} {y=}")
```

produces:

Root is approx. x=2.2748231592544493e-17 y=-1.0

Derivatives

- Motivation:
 - We need derivatives of functions for some optimisation and root finding algorithms
 - Not always is the function analytically known (but we are usually able to compute the function numerically)
 - The material presented here forms the basis of the finite-difference technique that is commonly used to solve ordinary and partial differential equations.
- The following slides show
 - the forward difference technique,
 - the backward difference technique and the
 - central difference technique to approximate the derivative of a function.
 - We also derive the accuracy of each of these methods.

The 1st derivative

- (Possible) Definition of the derivative (or “*differential operator*” $\frac{d}{dx}$)

$$\frac{df}{dx}(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- Use *difference operator* to approximate differential operator

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h}$$

- \Rightarrow can now compute *an approximation* of f' simply by evaluating f .
- This is called the *forward difference* because we use $f(x)$ and $f(x+h)$.
- Important question: How accurate is this approximation?

Example 1: forward difference

Using forward difference to estimate the derivative of $f(x) = \exp(x)$

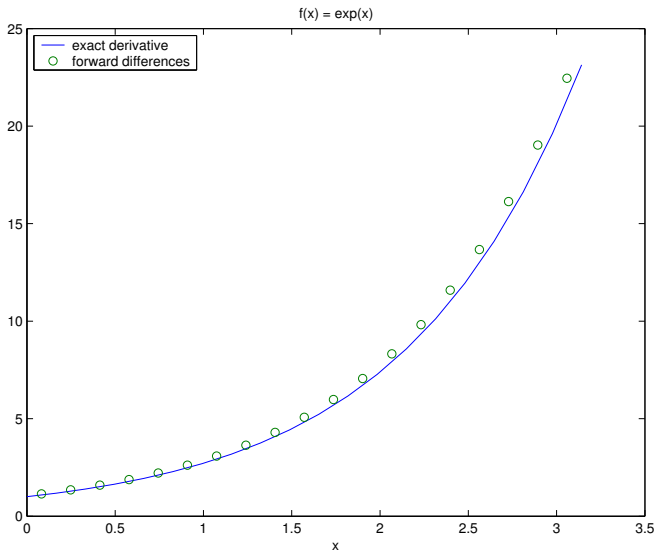
$$f'(x) \approx \frac{f(x+h) - f(x)}{h} = \frac{\exp(x+h) - \exp(x)}{h}$$

Numerical example:

- $h = 0.1, x = 1$
- $f'(1.0) \approx \frac{\exp(1.1) - \exp(1)}{0.1} = 2.8588$
- Exact answer is $f'(1.0) = \exp(1.0) = 2.7182$
- error is $2.8588 - 2.7182 = 0.1406$ (relative error is about 5%).

Example (1): forward difference

Comparison: forward difference and exact derivative of $\exp(x)$



Accuracy of the forward difference

- Formal derivation using the Taylor series of f around x

$$\begin{aligned}f(x+h) &= \sum_{n=0}^{\infty} h^n \frac{f^{(n)}(x)}{n!} \\ &= f(x) + hf'(x) + h^2 \frac{f''(x)}{2!} + h^3 \frac{f'''(x)}{3!} + \dots\end{aligned}$$

- Rearranging for $f'(x)$

$$\begin{aligned}hf'(x) &= f(x+h) - f(x) - h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!} - \dots \\ f'(x) &= \frac{1}{h} \left(f(x+h) - f(x) - h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!} - \dots \right) \\ &= \frac{f(x+h) - f(x)}{h} - \frac{h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!}}{h} - \dots \\ &= \frac{f(x+h) - f(x)}{h} - h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - \dots\end{aligned}$$

Accuracy of the forward difference (2)

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \underbrace{h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - \dots}_{E_{\text{forw}}(h)}$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} + E_{\text{forw}}(h)$$

- Therefore, the error term $E_{\text{forw}}(h)$ is

$$E_{\text{forw}}(h) = -h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - \dots$$

- Can also be expressed as

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

The 1st derivative using the backward difference

- Another definition of the derivative (or “differential operator” $\frac{d}{dx}$)

$$\frac{df}{dx}(x) = \lim_{h \rightarrow 0} \frac{f(x) - f(x - h)}{h}$$

- Use difference operator to approximate differential operator

$$\frac{df}{dx}(x) = \lim_{h \rightarrow 0} \frac{f(x) - f(x - h)}{h} \approx \frac{f(x) - f(x - h)}{h}$$

- This is called the *backward difference* because we use $f(x)$ and $f(x - h)$.
- How accurate is the backward difference?

Accuracy of the backward difference

- Formal derivation using the Taylor Series of f around x

$$f(x-h) = f(x) - hf'(x) + h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!} + \dots$$

- Rearranging for $f'(x)$

$$hf'(x) = f(x) - f(x-h) + h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!} - \dots$$

$$\begin{aligned} f'(x) &= \frac{1}{h} \left(f(x) - f(x-h) + h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!} - \dots \right) \\ &= \frac{f(x) - f(x-h)}{h} + \frac{h^2 \frac{f''(x)}{2!} - h^3 \frac{f'''(x)}{3!}}{h} - \dots \\ &= \frac{f(x) - f(x-h)}{h} + h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - \dots \end{aligned}$$

Accuracy of the backward difference (2)

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \underbrace{h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - \dots}_{E_{\text{back}}(h)}$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + E_{\text{back}}(h) \quad (7)$$

- Therefore, the error term $E_{\text{back}}(h)$ is

$$E_{\text{back}}(h) = h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - \dots$$

- Can also be expressed as

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h)$$

Combining backward and forward differences (1)

The approximations are

- forward:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + E_{\text{forw}}(h) \quad (8)$$

- backward

$$f'(x) = \frac{f(x) - f(x-h)}{h} + E_{\text{back}}(h) \quad (9)$$

$$E_{\text{forw}}(h) = -h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} - h^3 \frac{f^{(4)}(x)}{4!} - h^4 \frac{f^{(5)}(x)}{5!} - \dots$$

$$E_{\text{back}}(h) = h \frac{f''(x)}{2!} - h^2 \frac{f'''(x)}{3!} + h^3 \frac{f^{(4)}(x)}{4!} - h^4 \frac{f^{(5)}(x)}{5!} + \dots$$

⇒ Add equations (8) and (9) together, then the error cancels partly.

Combining backward and forward differences (2)

Add these lines together

$$f'(x) = \frac{f(x+h) - f(x)}{h} + E_{\text{forw}}(h)$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + E_{\text{back}}(h)$$

$$2f'(x) = \frac{f(x+h) - f(x-h)}{h} + E_{\text{forw}}(h) + E_{\text{back}}(h)$$

Adding the error terms:

$$E_{\text{forw}}(h) + E_{\text{back}}(h) = -2h^2 \frac{f'''(x)}{3!} - 2h^4 \frac{f''''(x)}{5!} - \dots$$

The combined (central) difference operator is

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + E_{\text{cent}}(h)$$

with

$$E_{\text{cent}}(h) = -h^2 \frac{f'''(x)}{3!} - h^4 \frac{f''''(x)}{5!} - \dots$$

Central difference

- Can be derived (as on previous slides) by adding forward and backward difference
- Can also be interpreted geometrically by defining the differential operator as

$$\frac{df}{dx}(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

and taking the finite difference form

$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Error of the central difference is only $\mathcal{O}(h^2)$, i.e. better than forward or backward difference

It is generally the case that symmetric differences are more accurate than asymmetric expressions.

Forward, backward and central differences

- Can approximate derivatives of f numerically
- need only function evaluations of f
- three different difference methods

name	formula	error
forward	$f'(x) = \frac{f(x+h)-f(x)}{h}$	$\mathcal{O}(h)$
backward	$f'(x) = \frac{f(x)-f(x-h)}{h}$	$\mathcal{O}(h)$
central	$f'(x) = \frac{f(x+h)-f(x-h)}{2h}$	$\mathcal{O}(h^2)$

Example 2: spacing h in central differences

```
def f(x):
    """Return x^3/3. (Derivative is x^2)."""
    return x**3 / 3

print("Change x, h=1e-6 is fixed =====")
h = 1e-6
print("      x      Numer.Derivative      abs. Error")
for x in range(0, 5):
    fprime = (f(x+h) - f(x-h)) / (2 * h)
    print(f"{x:8}  {fprime:20.15f}  {abs(fprime-x**2):10.6g}")

print("\nChange h, x=2 is fixed =====")
x = 2
print("      h      Numer.Derivative      abs. Error")
for h in [1e-1, 1e-3, 1e-6, 1e-7, 1e-9, 1e-12, 1e-15]:
    fprime = (f(x+h) - f(x-h)) / (2 * h)
    print(f"{h:8g}  {fprime:20.15f}  {abs(fprime-x**2):10.6g}")
```

Example 2 output

Change x, h=1e-6 is fixed =====

x	Numer.Derivative	abs. Error
0	0.0000000000000333	3.33333e-13
1	0.999999999973245	2.67555e-11
2	4.000000000115023	1.15023e-10
3	9.000000002146180	2.14618e-09
4	16.000000002236447	2.23645e-09

Change h, x=2 is fixed =====

h	Numer.Derivative	abs. Error
0.1	4.003333333333337	0.00333333
0.001	4.000000333332698	3.33333e-07
1e-06	4.000000000115023	1.15023e-10
1e-07	3.999999997894577	2.10542e-09
1e-09	4.000000330961484	3.30961e-07
1e-12	4.000355602329364	0.000355602
1e-15	3.996802888650563	0.00319711

- too large h : inaccurate approximation of derivative
- too small h : floating point representation errors

Summary

- Can approximate derivatives of f numerically
- need only function evaluations of f
- three different difference methods

name	formula	error
forward	$f'(x) = \frac{f(x+h)-f(x)}{h}$	$\mathcal{O}(h)$
backward	$f'(x) = \frac{f(x)-f(x-h)}{h}$	$\mathcal{O}(h)$
central	$f'(x) = \frac{f(x+h)-f(x-h)}{2h}$	$\mathcal{O}(h^2)$

- central difference is most accurate

* Note: Euler's (integration) method — derivation using finite difference operator

- Use forward difference operator to approximate differential operator

$$\frac{dy}{dx}(x) = \lim_{h \rightarrow 0} \frac{y(x+h) - y(x)}{h} \approx \frac{y(x+h) - y(x)}{h}$$

- Change differential to difference operator in $\frac{dy}{dx} = f(x, y)$

$$f(x, y) = \frac{dy}{dx} \approx \frac{y(x+h) - y(x)}{h}$$

$$hf(x, y) \approx y(x+h) - y(x)$$

$$\implies y_{i+1} = y_i + hf(x_i, y_i)$$

- \implies Euler's method (for ODEs) can be derived from the forward difference operator.

* Note: Newton's (root finding) method — derivation from Taylor series

- We are looking for a root, *i.e.* we are looking for a x so that $f(x) = 0$.
- We have an initial guess x_0 which we refine in subsequent iterations:

$$x_{i+1} = x_i - h_i \quad \text{where} \quad h_i = \frac{f(x_i)}{f'(x_i)}. \quad (10)$$

- This equation can be derived from the Taylor series of f around x . Suppose we guess the root to be at x and $x + h$ is the actual location of the root (so h is unknown and $f(x + h) = 0$):

$$\begin{aligned} f(x + h) &= f(x) + hf'(x) + \dots \\ 0 &= f(x) + hf'(x) + \dots \\ \implies 0 &\approx f(x) + hf'(x) \\ \iff h &\approx -\frac{f(x)}{f'(x)}. \end{aligned} \quad (11)$$

Numpy

numpy

- is an interface to high performance linear algebra libraries (such as BLAS, LAPACK, ATLAS, MKL, BLIS)
- provides
 - the `array` object (strictly `ndarray` type)
 - fast mathematical operations over arrays
 - linear algebra, Fourier transforms, random number generation
- Numpy is *not* part of the Python standard library.

numpy 1d-arrays (vectors)

- An (1d) array is a sequence of objects
- all objects in one array are of the same type

```
>>> import numpy as np # widely used convention
>>> a = np.array([1, 4, 10])
>>> a
array([ 1,  4, 10])
>>> type(a)
<class numpy.ndarray>
>>> a + 100
array([101, 104, 110])
>>> a**2
array([  1,  16, 100])
>>> np.sqrt(a)
array([ 1.          ,  2.          ,  3.16227766])
>>> a > 3
array([False,  True,  True], dtype=bool)
```

Array creation 1: from iterable

- 1d-array (vector) from iterable

```
>>> import numpy as np
>>> a = np.array([1, 4, 10]) # from list
>>> a
array([ 1,  4, 10])
>>> print(a)
[ 1  4 10]
```

- 2d-array (matrix) from nested sequences

```
>>> B = np.array([[0, 1.5], [10, 12]]) # from nested
>>> B
array([[ 0. ,  1.5],
       [10. , 12. ]])
>>> print(B)
[[ 0.  1.5]
 [10. 12. ]]
```

Array type

- All elements in an array must be of the same type
- For existing array, the type is the `dtype` attribute

```
>>> a.dtype  
dtype('int64')
```

```
>>> B.dtype  
dtype('float64')
```

- We can fix the type of the array when we create the array, for example:

```
>>> a2 = array([1, 4, 10], float)
```

```
>>> a2  
array([ 1.,  4., 10.]
```

```
>>> a2.dtype  
dtype('float64')
```

- For numerical calculations, we normally use double floats which are known as `float64` or short `float`:

```
>>> a2 = array([1, 4, 10], float)
```

```
>>> a2.dtype
```

```
dtype('float64')
```

- This is also the default type for `zeros` and `ones`.
- A full list is available at <http://docs.scipy.org/doc/numpy/user/basics.types.html>

Array size

The total number of elements is given through the `size` attribute:

```
>>> a.size
3
>>> B.size
4
```

The number of bytes per item:

```
>>> a.itemsize # dtype is int64
8
>>> B.itemsize # dtype is float64
8
```

The total number of bytes used is given through the `nbytes` attribute:

```
>>> a.nbytes
```

```
24
```

```
>>> B.nbytes
```

```
32
```

Diving in with numpy.info

```
>>> z = np.arange(0, 12, 1).reshape(3, 4)
>>> z
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> z.dtype
dtype('int64')
>>> np.info(z)
class: ndarray
shape: (3, 4)
strides: (32, 8) # 32 bytes from row to row
itemsize: 8
aligned: True
contiguous: True
fortran: False
data pointer: 0x6000012dc060
byteorder: little
byteswap: False
type: int64
>>> z.nbytes
96
```

Array creation 2: arange and linspace

- `arange([start,] stop[, step,])` is inspired by `range`: create array from `start` up to *but not including* `stop`

```
>>> np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.arange(10, dtype=float)
```

```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
>>> np.arange(0, 1, 0.1)
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

- `linspace(start, stop, num=50)` provides `num` points linearly spaced between `start` and `stop`:

```
>>> np.linspace(0, 10, 11)
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

```
>>> np.linspace(0, 1, 11)
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.])
```


Array shape

The shape is a tuple that describes

- (i) the dimensionality of the array (that is the length of the shape tuple) and
- (ii) the number of elements for each dimension (“axis”)

Example:

```
>>> a.shape
(3,)      # 1d array with 3 elements
>>> B.shape
(2, 2)    # 2d array with 2 x 2 elements
```

Can use shape attribute to change shape:

```
>>> B
array([[ 0. ,  1.5],
       [10. , 12. ]])
>>> B.shape
(2, 2)
>>> B.shape = (4,)
>>> B
array([ 0. ,  1.5, 10. , 12. ])
```

Number of dimension also available in attribute `ndim`:

```
>>> B.ndim
2
>>> len(B.shape) # same as B.ndim
2
```

Array indexing (1d arrays)

```
>>> x = np.arange(0, 10, 2)
>>> x
array([0, 2, 4, 6, 8])
>>> x[3]
6
>>> x[4]
8
>>> x[-1] # last element
8
```

Array indexing (2d arrays)

```
>>> C = np.arange(12)
>>> C
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> C.shape = (3, 4)
>>> C
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> C[0, 0] # first index for rows, second for columns
0
>>> C[2, 0]
8
>>> C[2, -1] # row 3, last column
11
>>> C[-1, -1] # last row, last column
11
```

Array slicing (1d arrays)

Double colon operator `::`

Read as `START:END:INDEXSTEP`

If either `START` or `END` are omitted, the respective ends of the array are used. `INDEXSTEP` defaults to 1.

Examples:

```
>>> y = np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y[0:5]           # slicing (default step is 1)
array([0, 1, 2, 3, 4])
>>> y[0:5:1]        # equivalent (step 1)
```

```
array([0, 1, 2, 3, 4])
>>> y[0:5:2]           # slicing with index step 2
array([0, 2, 4])
>>> y[:5:2]           # from the beginning
array([0, 2, 4])
>>> y[0:5:-1]         # negative index step size
array([], dtype=int64)
>>> y[5:0:-1]         # from end to beginning
array([5, 4, 3, 2, 1])
>>> y[5:0:-2]         # in steps of two
array([5, 3, 1])
>>> y[::-1]           # reverses array elements
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

[Double colon operator works for all sequences.]

Array slicing (2d)

Slicing for 2d (or higher dimensional arrays) is analog to 1-d slicing, but applied to each component. Common operations include extraction of a particular row or column from a matrix:

```
>>> C
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> C[0, :]          # row with index 0
array([0, 1, 2, 3])
>>> C[:, 1]         # column with index 1
                    # (i.e. 2nd col)
array([1, 5, 9])
```

Array creation 3: zeros and ones

Other useful methods are `zeros` and `ones` which accept a desired matrix shape as the input:

```
>>> np.zeros((2, 4))    # two rows, 4 cols
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.zeros((4,))     # (4,) is tuple
array([ 0.,  0.,  0.,  0.])
>>> np.zeros(4)        # 4 works as well
array([ 0.,  0.,  0.,  0.])

>>> np.ones((2, 7))
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.]])
```


Array creation 4: eye and diag

Create Identity matrix `eye` (name from capital I used in equations):

```
>>> np.eye(2)
array([[1., 0.],
       [0., 1.]])
```

Create diagonal matrix `diag`:

```
>>> np.diag([10, 20, 30])
array([[10,  0,  0],
       [ 0, 20,  0],
       [ 0,  0, 30]])
```

Views of numpy arrays

Slicing a numpy array results in a *view* of the data (not a copy).

```
>>> C = np.arange(12).reshape(3, 4)
>>> C
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> view_C = C[0, :]
>>> view_C
array([0, 1, 2, 3])
>>> C[0, 0] = 42
>>> view_C
array([42,  1,  2,  3])
```

Often, this is desired — in particular when the arrays are large.

array.base points to the view's data

- `x.base == None` means `x` is not a view.
- `x.base is y` means `x` is a view of `y`.

Example:

```
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print(x.base)
None
>>> y = x[::2] # create a view with every 2nd element
>>> print(y.base)
[0 1 2 3 4 5 6 7 8 9]
>>> y.base is x
True
>>> np.shares_memory(x, y) # check if x and y share mem
True
```

Creating copies of numpy arrays

Create copy of 1d array:

```
>>> y = np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> copy_y = y.copy()
>>> y[0] = 42
>>> copy_y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print(copy_y.base)
None
>>> np.shares_memory(y, copy_y)
False
```

Solving linear systems of equations

`np.linalg.solve(A, b)` solves $Ax = b$ for a square matrix A and given vector b , and returns the solution vector x .

Example:

$$Ax = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \end{pmatrix} = b$$

```
>>> A = np.array([[1, 0], [0, 2]])
>>> b = np.array([1, 4])
>>> from np.linalg import solve
>>> x = solve(A, b)
>>> x
array([ 1.,  2.])
>>> np.dot(A, x) # Computing A*x
array([ 1.,  4.]) # this should be b
```

`help(np.linalg)` provides an overview, including

- `det` to compute the determinant
- `eig` to compute eigenvalues and eigenvectors
- `pinv` to compute the (pseudo) inverse of a matrix
- `svd` to compute a singular value decomposition

- fast if number of elements is large: for an array with one element, `np.sqrt` will be slower than `math.sqrt`
- avoid loops (formulate instead as matrix operation)
- avoid copies of data (i.e. use views)
- numpy can be up to ~ 100 times faster than naive Python

- numpy provides fast array operations (comparable to Matlab's matrices)
- elements in the array have the same type (typically a numerical type)
- data is stored contiguously in memory (if possible)

- Consult **Numpy** documentation if used outside this course. Start here:
 - Basics: https://numpy.org/doc/stable/user/absolute_beginners.html
 - Quickstart: <https://numpy.org/doc/stable/user/quickstart.html>
- Matlab users may want to read **Numpy for Matlab Users**

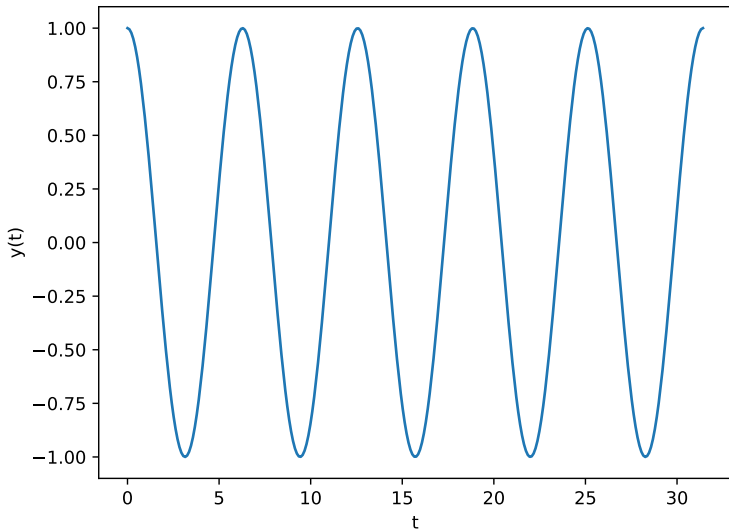
Preview: plotting data in arrays with matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

# compute data
t = np.linspace(0, 10 * np.pi, 300)
y = np.cos(t)

# create plot
fig, ax = plt.subplots() # default: one plot
ax.plot(t, y)
ax.set_xlabel("t")
ax.set_ylabel("y(t)")

# save figure to pdf
fig.savefig("plotting-array-matplotlib.pdf")
```



- Matplotlib tries to make easy things easy and hard things possible
- Matplotlib is a 2D plotting library which produces publication quality figures (increasingly also 3d)
- Matplotlib can be fully scripted but interactive interface available
- Two application interfaces: `pylab` and `matplotlib.pyplot`

Within the IPython console (for example in Spyder) and the Jupyter Notebook, use

- `%matplotlib inline` to see plots inside the console window, and
- `%matplotlib qt` to create pop-up windows with the plot. (May need to call `matplotlib.show()`.) We can manipulate the view interactively in that window.
- In Spyder, the plots appear by default in the “plots” pane.
- Within the Jupyter notebook, you can use `%matplotlib notebook` which embeds an interactive window in the note book.

Pylab

Pylab is a Matlab-like (state-driven) plotting interface (and comes with matplotlib).

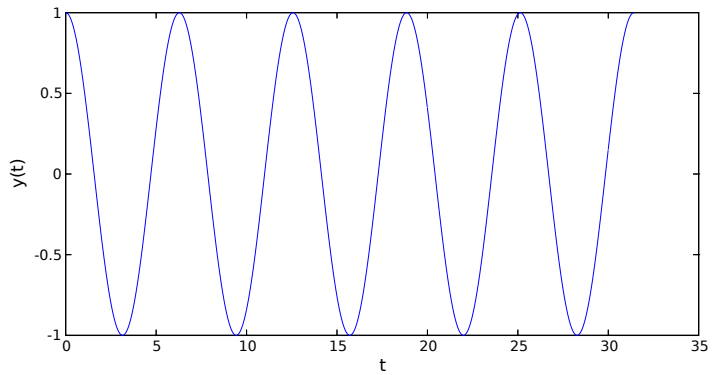
- Convenient for “simple and fast” plots. (In particular if you like Matlab.)
- Make use of `help(pylab.plot)` to remind you of line styles, symbols etc.

Plotting arrays with pylab (Matlab style)

```
# Plot example (Matlab style)
import pylab
import numpy as np

t = np.arange(0, 10 * np.pi, 0.01)
y = np.cos(t)

pylab.plot(t, y)
pylab.xlabel("t")
pylab.ylabel("y(t)")
pylab.show()
```



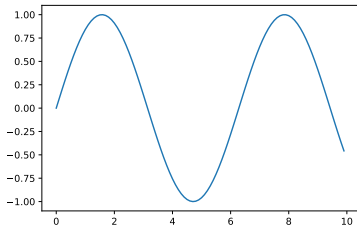
- **Matplotlib.pyplot** is an object oriented plotting interface
- Very fine grained control over plots
- recommended to use

matplotlib.pyplot - example 1

```
import math
import matplotlib.pyplot as plt

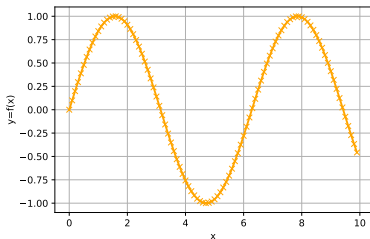
# create some data
xs = [0.1*xi for xi in range(100)]
ys = [math.sin(x) for x in xs]

# create plot
fig, ax = plt.subplots()
ax.plot(xs, ys)
```



matplotlib.pyplot - example 2

```
fig, ax = plt.subplots(figsize=(6, 4))  
ax.plot(xs, ys, 'x-', linewidth=2, color='orange')  
  
ax.grid('on')  
ax.set_xlabel('x')  
ax.set_ylabel('y=f(x)')  
fig.savefig("pyplot-demo2.pdf")
```



Matplotlib.pyplot

Matplotlib.pyplot is an object oriented plotting interface.

- prefer this over `pylab`
- Check gallery at <https://matplotlib.org/stable/gallery/index.html>
- Try [Matplotlib notebook \(on module's home page\)](#) as an introduction and useful reference
- Nicolas Rougier. Scientific Visualization: Python + Matplotlib. Nicolas P. Rougier. 2021, 978-2- 9579901-0-8. hal-03427242, online at <https://github.com/rougier/scientific-visualization-book>

Higher Order Functions 2: Functional tools

- So far, have processed lists by iterating through them using for-loop
- perceived to be conceptually simple (by most learners) but
- not as compact as possible and not always as fast as possible
- Alternatives:
 - list comprehension
 - `map`, `filter`, `reduce`, often used with `lambda`

Anonymous function lambda

- lambda: anonymous function (function literal)
- Useful to define a small helper function that is only needed once

```
>>> lambda a: a
<function <lambda> at 0x319c70>
>>> lambda a: 2 * a
<function <lambda> at 0x319af0>
>>> (lambda a: 2 * a)
<function <lambda> at 0x319c70>
>>> (lambda a: 2 * a)(10)
20
>>> (lambda a: 2 * a)(20)
40
```

```
>>> (lambda x, y: x + y)(10, 20)
30
>>> (lambda x, y, z: (x + y) * z )(10, 20, 2)
60
>>> type(lambda x, y: x + y)
<type 'function'>
```


Lambda usage example 1

Integrate $f(x) = x^2$ from 0 to 2 (numerically):

- Without lambda (lambda1.py):

```
from scipy.integrate import quad
def f(x):
    return x * x
```

```
y, abserr = quad(f, a=0, b=2)
print("value is {y:f} +- {abserr:g}")
```

- With lambda (lambda1b.py):

```
from scipy.integrate import quad
y, abserr = quad(lambda x: x * x, a=0, b=2)
print("value is {y:f} +- {abserr:g}")
```

Output (same for both programs):

```
value is 2.666667 +- 2.96059e-14
```

Higher order functions

Roughly: “Functions that take or return functions” (see for example [Wikipedia entry](#))

Rough summary (check `help(COMMAND)` for details)

- `map(function, iterable) → iterable`:
apply function to all elements in iterable
- `filter(function, iterable) → iterable`:
return items of iterable for which `function(item)` is true.
- `reduce(function, iterable, initial) → value`:
apply `function(x,y)` from left to right to reduce iterable to a single value.

Note that sequences are iterables.

- `map(function, iterable) → iterable`: apply function to all elements in sequence

- Example:

```
>>> def f(x):
...     return x ** 2
>>> map(f, [0, 1, 2, 3, 4])
<map object at 0x1026a52e8>           # this is iterable
>>> list(map(f, [0, 1, 2, 3, 4]))    # convert to list
[0, 1, 4, 9, 16]
```

- `lambda` converts an expression (`x ** 2`) to a function:

```
>>> list(map(lambda x: x ** 2, [0, 1, 2, 3, 4]))
[0, 1, 4, 9, 16]
```

- Equivalent operation using list comprehension:

```
>>> [x ** 2 for x in [0, 1, 2, 3, 4]]
[0, 1, 4, 9, 16]
```

Examples map

- Example (maths):

```
>>> import math
>>> list(map(math.exp, [0, 0.1, 1.]))
[1.0, 1.1051709180756477, 2.718281828459045]
```

- Example (slug):

```
>>> news="Python programming occasionally \
... more fun than expected"
>>> slug = "-".join(map(
... lambda w: w[0:6], news.split()))
>>> slug
'Python-progra-occasi-more-fun-than-expect'
```

Equivalent list comprehension expression:

```
>>> slug = "-".join([w[0:6] for w in news.split()])
```

`filter(function, iterable) → iterable`: return items of iterable for which `function(item)` is true:

```
>>> def is_positive(n): # returns True for positive n
...     return n > 0
>>> list(filter(is_positive,
...             [-3, -2, -1, 0, 1, 2, 3, 4]))
[1, 2, 3, 4]
>>> list(filter(lambda n: n > 0,
...             [-3, -2, -1, 0, 1, 2, 3, 4]))
[1, 2, 3, 4]
```

List comprehension equivalent:

```
>>> [n for n in [-3, -2, -1, 0, 1, 2, 3, 4] if n > 0]
[1, 2, 3, 4]
```

Examples filter

```
>>> c = "The quick brown fox jumps".split()
>>> print(c)
['The', 'quick', 'brown', 'fox', 'jumps']
>>> def len_gr_4(s): # return True if s has >4 letters
...     return len(s) > 4
>>> list(map(len_gr_4, c))
[False, True, True, False, True]
>>> filter(len_gr_4, c)
<filter object at 0x10522e5c0>
>>> list(filter(len_gr_4, c))
['quick', 'brown', 'jumps']
>>> list(filter(lambda s: len(s) > 4, c))
['quick', 'brown', 'jumps']
```

Equivalent operation using list comprehension:

```
>>> [s for s in c if len(s) > 4]
['quick', 'brown', 'jumps']
```

Reduce

- `functools.reduce(function, iterable, initial) → value:`
apply `function(x, y)` from left to right to reduce iterable to a single value.
- Examples:

```
>>> from functools import reduce
>>> def f(x, y):
...     print(f"Called with {x=}, {y=}")
...     return x + y
...
>>> reduce(f, [1, 3, 5], 0)
Called with x=0, y=1
Called with x=1, y=3
Called with x=4, y=5
9
```



```
>>> reduce(f, [1, 3, 5], 100)
Called with x=100, y=1
Called with x=101, y=3
Called with x=104, y=5
109
>>> reduce(f, "test", "")
Called with x=, y=t
Called with x=t, y=e
Called with x=te, y=s
Called with x=tes, y=t
'test'
>>> reduce(f, "test", "FIRST")
Called with x=FIRST, y=t
Called with x=FIRSTt, y=e
Called with x=FIRSTte, y=s
Called with x=FIRSTtes, y=t
'FIRSTtest'
```

*Operator module

- operator module contains functions which are typically accessed not by name, but via some symbols or special syntax.
- For example `3 + 4` is equivalent to `operator.add(3, 4)`. Thus:

```
def f(x, y): return x + y
reduce(f, range(10), 0)
```

can also be written as:

```
reduce(operator.add, range(10), 0)
```

Note: could also use:

```
reduce(lambda x, y: x + y, range(10), 0)
```

but use of `operator` module is preferred (often faster).

- Functions like **map**, **reduce** and **filter** are found in just about any language supporting functional programming.
- provide functional abstraction for commonly written loops
- Use those (and/or list comprehension) instead of writing loops, because
 - Writing loops by hand is quite tedious and error-prone.
 - The functional version is often clearer to read.
 - The functional version can result in faster code (if you can avoid **lambda**)

What command to use when?

- **lambda** allows to define a (usually simple) function "in-place". We need this to convert an *expression* into a *function*.
- **map** transforms a sequence to another sequence (of same length) using a function
- **filter** filters a sequence (reduces number of elements) using a function
- **list comprehension** transforms a list (can include filtering) using an expression
 - if you need to use a **lambda** in a **map**, you are probably better off using list comprehension.
 - if you have a function to apply, **map** is more compact than a list comprehension.
- **reduce** carries out an operation that "collects" information (sum, product, ...), for example reducing the sequence to a single number.

Example: squaring elements in list with expression `x**2`

Some alternatives:

```
>>> res = []
>>> for x in range(5):
...     res.append(x ** 2)
... 
```

```
>>> res
[0, 1, 4, 9, 16]
```

```
>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]
```

```
>>> list(map(lambda x: x ** 2, range(5)))
[0, 1, 4, 9, 16]
```

Example: squaring elements in list with function f

```
>>> def f(x):  
...     return x**2  
  
>>> res = []  
>>> for x in range(5):  
...     res.append(f(x))  
...  
>>> res  
[0, 1, 4, 9, 16]  
  
>>> [f(x) for x in range(5)]  
[0, 1, 4, 9, 16]  
  
>>> list(map(f, range(5)))  
[0, 1, 4, 9, 16]
```


Numerical Integration

Different situations where we use integration:

(A) solving (definite) integrals

(B) solving (ordinary) differential equations

- more complicated than (A)
- Euler's method, Runge-Kutta methods

Both (A) and (B) are important.

We begin with the numeric computation of integrals (A).

(A) Definite Integrals

Often written as

$$I = \int_a^b f(x) dx \quad (12)$$

- example: $I = \int_0^2 \exp(-x^2) dx$
- solution is $I \in \mathbb{R}$ (i.e. a number)
- right hand side $f(x)$ depends only on x
- if $f(x) > 0 \quad \forall x \in [a, b]$, then we can visualise I as the area underneath $f(x)$
 - Note that the integral is *not* necessarily the same as the area enclosed by $f(x)$ and the x -axis:
 - $\int_0^{2\pi} \sin(x) dx = 0$
 - $\int_0^1 (-1) dx = -1$

(B) Ordinary Differential Equations (ODE)

Often written as

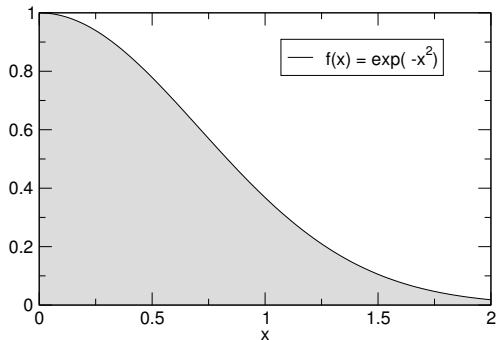
$$y' \equiv \frac{dy}{dx} = f(x, y) \quad (13)$$

- example: $\frac{dv}{dt} = \frac{1}{m}(g - cv^2)$
- solution is $y(x) : \mathbb{R} \rightarrow \mathbb{R}$ (i.e. a function)
 $x \mapsto y(x)$
- right hand side $f(x, y)$ depends on x and on solution y
- Can write (13) formally as $y = \int \frac{dy}{dx} dx = \int f(x, y) dx$. That's why we "integrate differential equations" to solve them.

Numeric computation of definite integrals

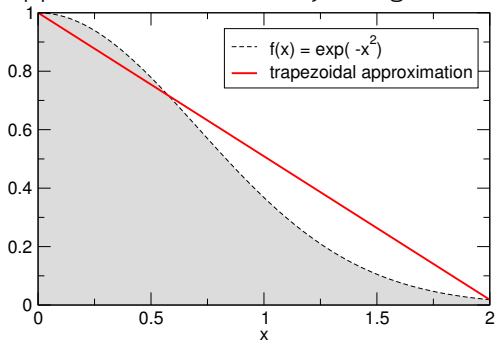
Example:

$$I = \int_0^2 \exp(-x^2) dx$$



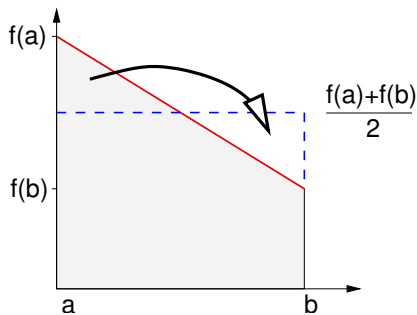
Simple trapezoidal rule

- Approximate function by straight line



Simple trapezoidal rule (2)

- Compute area underneath straight line $p(x)$



- Result

$$A = \int_a^b p(x) dx = (b - a) \frac{f(a) + f(b)}{2}$$

Simple trapezoidal rule (3)

Aim: compute

$$I = \int_a^b f(x) dx$$

Strategy:

- approximate $f(x)$ with a linear function $p(x)$:

$$p(x) \approx f(x)$$

- compute the area A underneath that function $p(x)$:

$$A = \int_a^b p(x) dx = (b - a) \frac{f(a) + f(b)}{2}$$

- approximate

$$I = \int_a^b f(x) dx \approx \int_a^b p(x) dx = A = (b - a) \frac{f(a) + f(b)}{2}$$

Simple trapezoidal rule (4) Example

- Integrate $f(x) = x^2$

$$I = \int_0^2 x^2 dx$$

- What is the (correct) analytical answer?

Integrating polynomials:

$$I = \int_a^b x^k dx = \left[\frac{1}{k+1} x^{k+1} \right]_a^b$$

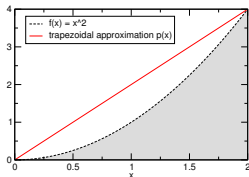
- for $a = 0$ and $b = 2$ and $k = 2$

$$I = \left[\frac{1}{2+1} x^{2+1} \right]_0^2 = \frac{1}{3} 2^3 = \frac{8}{3} \approx 2.6667$$

- Using the trapezoidal rule

$$A = (b - a) \frac{f(a) + f(b)}{2} = 2 \frac{0 + 4}{2} = 4$$

- The correct answer is $I = 8/3$ and the approximation is $A = 4$.
We thus *overestimate* I by $\frac{A-I}{I} \approx 50\%$.
- Plotting $f(x) = x^2$ together with the approximation reveals why we overestimate I



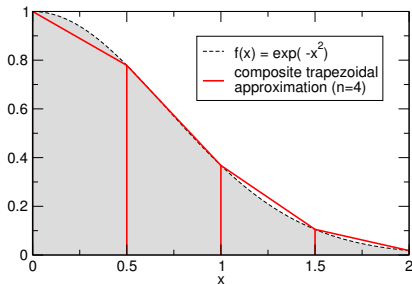
- The linear approximation, $p(x)$, overestimates $f(x)$ everywhere (except at $x = a$ and $x = b$).
Therefore, the integral of $p(x)$ is greater than the integral of $f(x)$.

(More formally: $f(x)$ is convex on $[a, b] \iff f''(x) \geq 0 \quad \forall x \in [a, b]$.)

Composite trapezoidal rule

Example $f(x) = \exp(-x^2)$:

$$I = \int_0^2 f(x) dx = \int_0^2 \exp(-x^2) dx$$



$$I = \int_0^{0.5} f(x) dx + \int_{0.5}^1 f(x) dx + \int_1^{1.5} f(x) dx + \int_{1.5}^2 f(x) dx$$

General composite trapezoidal rule

For n subintervals the formulae for the composite trapezoidal rule are

$$\begin{aligned}h &= \frac{b-a}{n} \\x_i &= a + ih \quad \text{with } i = 1, \dots, n-1 \\A &= \frac{h}{2} \left(f(a) + 2f(x_1) + 2f(x_2) + \dots \right. \\&\quad \left. + 2f(x_{n-2}) + 2f(x_{n-1}) + f(b) \right) \\&= \frac{h}{2} \left(f(a) + \sum_{i=1}^{n-1} 2f(x_i) + f(b) \right)\end{aligned}$$

Error of composite trapezoidal rule

One of the important (and difficult) questions in numerical analysis and computing is:

- How accurate is my approximation?

For integration methods, we are interested in how much the error decreases when we decrease h (by increasing the number of subintervals, n).

For the composite trapezoidal rule it can be shown that:

$$\int_a^b f(x) dx = \frac{h}{2} \left(f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right) + \mathcal{O}(h^2)$$

The symbol $\mathcal{O}(h^2)$ means that the error term is (smaller or equal to an upper bound which is) proportional to h^2 :

- If we take 10 times as many subintervals then h becomes 10 times smaller (because $h = \frac{b-a}{n}$) and the error becomes 100 times smaller (because $\frac{1}{10^2} = \frac{1}{100}$).

Error of composite trapezoidal rule, example

- The table below shows how the error of the approximation, A , decreases with increasing n for

$$I = \int_0^2 x^2 dx.$$

n	h	A	I	$\Delta = A - I$	rel.err. = Δ / I
1	2.000000	4.000000	2.666667	1.333333	50.0000%
2	1.000000	3.000000	2.666667	0.333333	12.5000%
3	0.666667	2.814815	2.666667	0.148148	5.5556%
4	0.500000	2.750000	2.666667	0.083333	3.1250%
5	0.400000	2.720000	2.666667	0.053333	2.0000%
6	0.333333	2.703704	2.666667	0.037037	1.3889%
7	0.285714	2.693878	2.666667	0.027211	1.0204%
8	0.250000	2.687500	2.666667	0.020833	0.7813%
9	0.222222	2.683128	2.666667	0.016461	0.6173%
10	0.200000	2.680000	2.666667	0.013333	0.5000%
50	0.040000	2.667200	2.666667	0.000533	0.0200%
100	0.020000	2.666800	2.666667	0.000133	0.0050%

- The accuracy we actually require depends on the problem under investigation – no general statement is possible.

Summary trapezoidal rule for numerical integration

- Aim: to find an approximation of

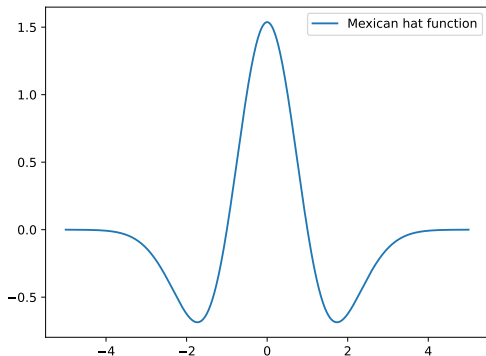
$$I = \int_a^b f(x) dx$$

- Simple trapezoidal method:
 - approximate $f(x)$ by a simpler (linear) function $p(x)$ and
 - integrate the approximation $p(x)$ exactly.
- Composite trapezoidal method:
 - divides the interval $[a, b]$ into n equal subintervals
 - employs the simple trapezoidal method for each subinterval
 - has an error term of order h^2 .

Numpy usage examples

Making calculations fast with numpy

- Calculations using numpy are faster (~ 100 times) than using pure Python (see example next slide).
- Imagine we need to compute the mexican hat function with many points



Making calculations fast with numpy

```
"""Demo: practical use of numpy (mexhat-numpy.py)"""
import time
import math
import matplotlib.pyplot as plt
import numpy as np

N = 100000

def mexhat_py(t, sigma=1):
    """Computes Mexican hat shape, see
    http://en.wikipedia.org/wiki/Mexican\_hat\_wavelet for
    equation (13 Dec 2011)"""
    c = 2.0 / math.sqrt(3 * sigma) * math.pi**0.25
    return c * (1 - t**2 / sigma**2) * math.exp(
        -(t**2) / (2 * sigma**2))

def mexhat_np(t, sigma=1):
```

```
"""Computes Mexican hat shape using numpy, see
http://en.wikipedia.org/wiki/Mexican\_hat\_wavelet for
equation (13 Dec 2011)"""
```

```
c = 2.0 / math.sqrt(3 * sigma) * math.pi**0.25
return c * (1 - t**2 / sigma**2) * np.exp(
    -(t**2) / (2 * sigma**2))
```

```
def test_is_really_the_same():
```

```
    """Checking whether mexhat_np and mexhat_py produce
    the same results."""
```

```
    xs1, ys1 = loop1()
```

```
    xs2, ys2 = loop2()
```

```
    deviation = math.sqrt(sum((ys1 - ys2) ** 2))
```

```
    print("error:", deviation)
```

```
    assert deviation < 1e-14
```

```
def loop1():
```

```
    """Compute arrays xs and ys with mexican hat function
    in ys(xs), returns tuple (xs,ys)"""
```

```
    xs = np.linspace(-5, 5, N)
```

```
    ys = []
```

```
for x in xs:
    ys.append(mexhat_py(x))
return xs, ys

def loop2():
    """As loop1, but uses numpy to be faster."""
    xs = np.linspace(-5, 5, N)
    return xs, mexhat_np(xs)

def time_this(f):
    """Call f, measure and return number of seconds
    execution of f() takes"""
    starttime = time.time()
    f()
    stoptime = time.time()
    return stoptime - starttime

def make_plot(filenameroot):
    fig, ax = plt.subplots()
    xs, ys = loop2()
    ax.plot(xs, ys, label="Mexican hat function")
```

```
ax.legend()
fig.savefig(filenameroot + ".png")
fig.savefig(filenameroot + ".pdf")

def main():
    test_is_really_the_same()
    make_plot("mexhat1d")
    time1 = time_this(loop1)
    time2 = time_this(loop2)
    print(f"Numpy version is {time1 / time2:.1f} times faster")

if __name__ == "__main__":
    main()
```

Produces this output:

```
error: 1.1410712297602934e-15
Numpy version is 119.8 times faster
```

A lot of the source code above is focussed on measuring the execution time.

Within IPython, we could just have used `%timeit loop1()` and `%timeit loop2()` to get to the same timing information.

arrays with only one item convert to python scalars

```
>>> b = np.array([4])
>>> b.shape
(1,)
>>> type(b)
numpy.ndarray
>>> float(b)
4.0
>>> a = np.array([4])
>>> type(a)
numpy.ndarray
>>> a.shape
(1,)
>>> float(a)
```

4.0

```
>>> import math; math.sqrt(a)
```

2.0

This allows us to write functions $f(x)$ that can take an input argument x which can either be a `numpy.array` or a scalar. The `mexhat_np(t)` function is such an example:

```
>>> a = mexhat_np(0); print(f"{a=}, {type(a)=}")
a=1.537293661343647, type(a)=<class 'numpy.float64'>
```

```
>>> a = mexhat_np(np.array([0])); print(f"{a=}, {type(a)=}")
a=array([1.53729366]), type(a)=<class 'numpy.ndarray'>
```

```
>>> a = mexhat_np(np.linspace(0, 1, 3)); print(f"{a=}, {type(a)=}")
a=array([1.53729366, 1.01749267, 0.]), type(a)=<class 'numpy.ndarray'>
```


Closures

Returning function objects

We have seen that we can pass function objects as arguments to a function. Now we look at functions that *return function objects*.

Example (`closure_adder42.py`):

```
def make_add42():  
    def add42(x):  
        return x + 42  
    return add42
```

```
add42 = make_add42()  
print(add42(2))           # output is '44'
```

Closures

A closure ([Wikipedia](#)) is a function with bound variables. We often create closures by calling a function that returns a (specialised) function. For example (`closure_adder.py`):

```
import math

def make_adder(y):
    def adder(x):
        return x + y
    return adder

add42 = make_adder(42)
addpi = make_adder(math.pi)
print(add42(2))           # output is 44
print(addpi(-3))         # output is 0.14159265359
```

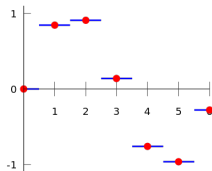
Scientific Python

(Partial) output of `help(scipy)`:

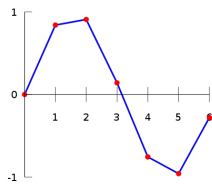
```
cluster      --- Vector Quantization / Kmeans
fft          --- Discrete Fourier transforms
fftpack     --- Legacy discrete Fourier transforms
integrate    --- Integration routines
interpolate  --- Interpolation Tools
io           --- Data input and output
linalg      --- Linear algebra routines
linalg.blas --- Wrappers to BLAS library
linalg.lapack --- Wrappers to LAPACK library
misc        --- Various utilities that don't have
              another home.
ndimage     --- N-D image package
odr         --- Orthogonal Distance Regression
```

optimize --- Optimization Tools
signal --- Signal Processing Tools
signal.window --- Window functions
sparse --- Sparse Matrices
sparse.linalg --- Sparse Linear Algebra
spatial --- Spatial data structures and algorithms
special --- Special functions
stats --- Statistical Functions

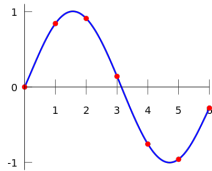
Interpolation of discrete data points



Piecewise constant interpolation



Linear interpolation



Polynomial interpolation

Interpolation of data

Given a set of N points (x_i, y_i) with $i = 1, 2, \dots, N$, we sometimes need a function $f(x)$ which returns $y_i = f(x_i)$ and interpolates the data between the x_i .

- `→ y0 = scipy.interpolate.interp1d(x, y)`
provides this interpolation
- `interp1d` returns a callable `y0` which interpolates the x - y data for any given x when called as `y0(x)`.
- Data interpolation of $y_i = f(x_i)$ may be useful to
 - create smoother plots of $f(x)$
 - find minima/maxima of $f(x)$
 - find x_c so that $f(x_c) = y_c$, provide inverse function $x = f^{-1}(y)$
 - integrate $f(x)$

Interpolation example

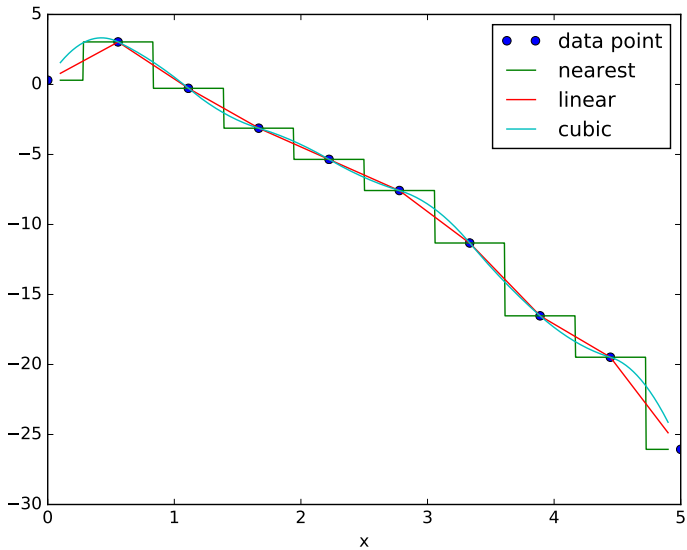
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate

def create_data(n):
    """Given an integer n, returns n data points
    x and values y as a numpy.array."""
    xmax = 5.0
    x = np.linspace(0, xmax, n)
    y = -(x ** 2)
    # make x-data somewhat irregular
    y += 1.5 * np.random.normal(size=len(x))
    return x, y

# main program
n = 10
x, y = create_data(n)
```

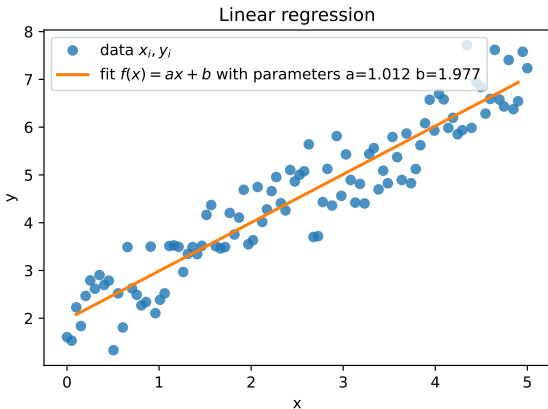
```
# use finer and regular mesh for plot
xfine = np.linspace(0.1, 4.9, n * 100)
# interpolate with piecewise constant function (p=0)
y0 = scipy.interpolate.interp1d(x, y, kind="nearest")
# interpolate with piecewise linear func (p=1)
y1 = scipy.interpolate.interp1d(x, y, kind="linear")
# interpolate with cubic spline
y2 = scipy.interpolate.interp1d(x, y, kind="cubic")

fig, ax = plt.subplots()
ax.plot(x, y, "o", label="data point")
ax.plot(xfine, y0(xfine), label="nearest")
ax.plot(xfine, y1(xfine), label="linear")
ax.plot(xfine, y2(xfine), label="cubic")
ax.legend()
ax.set_xlabel("x")
fig.savefig("interpolate.pdf")
```



Curve fitting

Given n data points $(x_i, y_i), i = 1, \dots, n$, and a model $y = f(x, \vec{p})$, with model parameters $\vec{p} = (p_1, p_2, \dots)$, find coefficients \vec{p} so that $y_i = f(x_i, \vec{p})$ describes the data “best”.



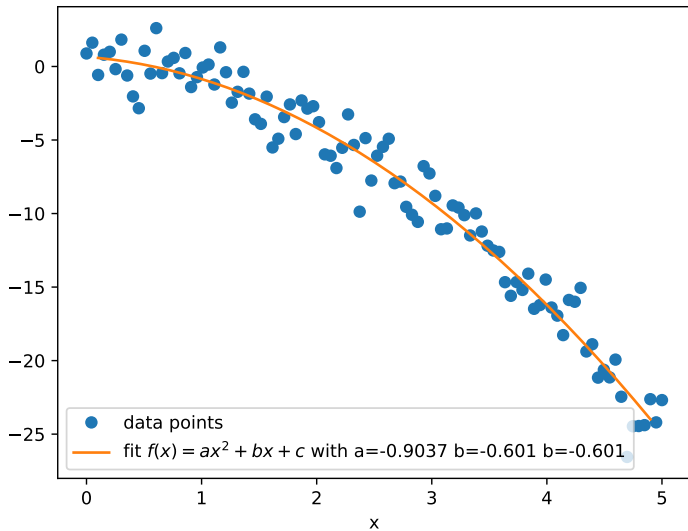
Curve fitting example

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize

def create_data(n):
    """Given an integer n, returns n data points
    x and values y as a numpy.array."""
    xmax = 5.0
    x = np.linspace(0, xmax, n)
    y = -x**2
    # make x-data somewhat irregular
    y += 1.5 * np.random.normal(size=len(x))
    return x, y

def model(x, a, b, c): # Equation for fit
    return a * x ** 2 + b * x + c
```

```
# main program
n = 100
x, y = create_data(n)
# do curve fit
p, pcov = scipy.optimize.curve_fit(model, x, y)
a, b, c = p
# plot fit and data
xfine = np.linspace(0.1, 4.9, n * 5)
fig, ax = plt.subplots()
ax.plot(x, y, "o", label="data points")
label = fr"fit  $f(x) = ax^2 + bx + c$  with {a:.4} {b:.4} {b:.4}"
ax.plot(xfine, model(xfine, a, b, c), label=label)
ax.legend()
ax.set_xlabel("x")
fig.savefig("curvefit2.pdf")
```



Function integration example

Aim: Compute $I = \int_a^b f(x)dx$, with
 $a = -2, b = 2, f(x) = \exp(-\cos(2x\pi)) + 3.2$

```
from math import cos, exp, pi
from scipy.integrate import quad

# function we want to integrate
def f(x):
    return exp(cos(-2 * x * pi)) + 3.2

# call quad to integrate f from -2 to 2
res, err = quad(f, -2, 2)

print(f"The numerical result is {res:f} (+-{err:g})")
```

which produces this output:

```
The numerical result is 17.864264 (+-1.55117e-11)
```

LAB9, 10

Optimisation (Minimisation)

- Optimisation typically described as:
given a function $f(x)$, find x_m so that $f(x_m)$ is the (local) minimum of f .
- To maximise $f(x)$, create a second function $g(x) = -f(x)$ and minimise $g(x)$.
- Optimisation algorithms need to be given a starting point (initial guess x_0 as close as possible to x_m)
- Minimum position x obtained may be local (not global) minimum

Optimisation example

```
import numpy as np
from scipy.optimize import fmin
import matplotlib.pyplot as plt

def f(x):
    return np.cos(x) - 3 * np.exp(-((x - 0.2) ** 2))

# find minima of f(x),
# starting from 1.0 and 2.0 respectively
minimum1 = fmin(f, 1.0)
print("Start search at x=1., minimum is", minimum1)
minimum2 = fmin(f, 2.0)
print("Start search at x=2., minimum is", minimum2)

# plot function
```

```
x = np.arange(-10, 10, 0.1)
y = f(x)
fig, ax = plt.subplots()
ax.plot(x, y, label=r"$\cos(x)-3e^{-(x-0.2)^2}$")
ax.set_xlabel("$x$")
ax.grid()
ax.axis([-5, 5, -2.2, 0.5])

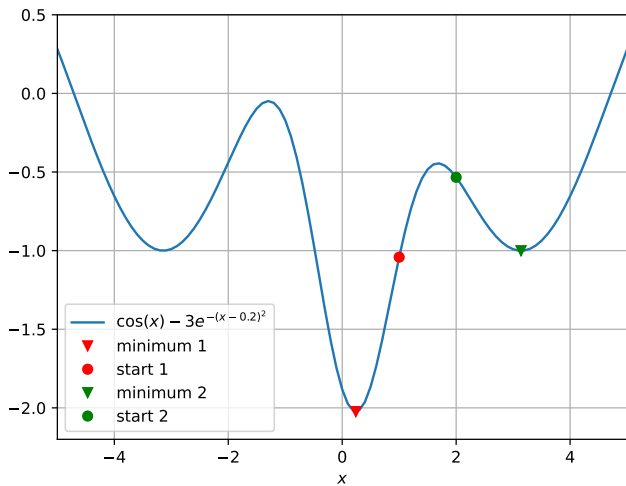
# add minimum1 to plot
ax.plot(minimum1, f(minimum1), "vr", label="minimum 1")
# add start1 to plot
ax.plot(1.0, f(1.0), "or", label="start 1")

# add minimum2 to plot
ax.plot(minimum2, f(minimum2), "vg", label="minimum 2")
# add start2 to plot
ax.plot(2.0, f(2.0), "og", label="start 2")
```

```
ax.legend(loc="lower left")  
fig.savefig("fmin1.pdf")
```

Code produces this output:

```
Optimization terminated successfully.  
    Current function value: -2.023866  
    Iterations: 16  
    Function evaluations: 32  
Start search at x=1., minimum is [ 0.23964844]  
Optimization terminated successfully.  
    Current function value: -1.000529  
    Iterations: 16  
    Function evaluations: 32  
Start search at x=2., minimum is [ 3.13847656]
```



LAB11

FIFO example and Object Oriented Programming (OOP)

Object Orientation (OO) and Closures

Earlier, we did an exercise for a first-in-first-out queue. At the time, we used a global variable to keep the state of the queue. To compare different approaches, the following slides show:

1. the original FIFO-queue solution (using a global variable, generally not good)
2. a modified version where the queue variable is passed to every function (→ this is object oriented programming without objects)
3. an object oriented version (where the queue data is part of the queue object). Probably the best solution, see OO programming for details.
4. a version based on closures (where the state is part of the closures)

Original FIFO solution (fifoqueue.py)

```
queue = []
def length():
    """Returns number of waiting customers"""
    return len(queue)

def show():
    """print list of customers, longest waiting customer at end."""
    for name in queue:
        print(f"waiting customer: {name}")

def add(name):
    """Customer with name 'name' joining the queue"""
    queue.insert(0, name)

def next():
    """Returns name of next to serve, removes customer from queue"""
    return queue.pop()

add('Spearing'); add('Fangoehr'); add('Takeda')
show(); next()
```


Improved FIFO solution

Improved FIFO solution (fifoqueue2.py)

```
def length(queue):
    return len(queue)

def show(queue):
    for name in queue:
        print(f"waiting customer: {name}")

def add(queue, name):
    queue.insert(0, name)

def next(queue):
    return queue.pop()

q1 = []
q2 = []
add(q1, 'Spearing'); add(q1, 'Fangohr'); add(q1, 'Takeda')
add(q2, 'John'); add(q2, 'Peter')
print(f"{length(q1)} customers in queue1:"); show(q1)
print(f"{length(q2)} customers in queue2:"); show(q2)
```

Object-Oriented FIFO solution (fifoqueue00.py)

```
class Fifoqueue:
    def __init__(self):
        self.queue = []

    def length(self):
        return len(self.queue)

    def show(self):
        for name in self.queue:
            print(f"waiting customer: {name}")

    def add(self, name):
        self.queue.insert(0, name)

    def next(self):
        return self.queue.pop()

q1 = Fifoqueue(); q2 = Fifoqueue()
q1.add('Spearing'); q1.add('Fangohr'); q1.add('Takeda')
q2.add('John'); q2.add('Peter')
print(f"{q1.length()} customers in queue1:"); q1.show()
```

*Functional (closure) FIFO solution (fifoqueue_closure.py)

```
def make_queue():  
    queue = []  
    def length():  
        return len(queue)  
  
    def show():  
        for name in queue: print(f"waiting customer: {name}")  
  
    def add(name):  
        queue.insert(0, name)  
  
    def next():  
        return queue.pop()  
    return add, next, show, length
```

```
q1_add, q1_next, q1_show, q1_length = make_queue()  
q2_add, q2_next, q2_show, q2_length = make_queue()  
q1_add('Spearing'); q1_add('Fangohr'); q1_add('Takeda')  
q2_add('John'); q2_add('Peter')  
print(f"{q1_length()} customers in queue1:"); q1_show()  
print(f"{q2_length()} customers in queue2:"); q2_show()
```

*Advanced: Using double-ended-queue (deque)

Specialised *double-ended-queue* data structure deque [1] available in the Collections module of python:

```
from collections import deque

def length(queue):
    return len(queue)

def show(queue):
    for name in queue:
        print(f"waiting customer: {name}")

def add(queue, name):
    queue.appendleft(name)

def next(queue):
    return queue.pop()

q1 = deque()
add(q1, 'Spearing'); add(q1, 'Fangohr'); add(q1, 'Takeda')
```

[1] <https://docs.python.org/3/library/collections.html#collections.deque>

Object orientation (OO):

- one important idea is to combine data and functions operating on data (in objects),
- objects contain data but
- access to data through interface (implementation details irrelevant to user)
- can program in OO style without OO-programming language:
 - as in FIFO2 solution
 - as in closure based approach
- OO mainstream programming paradigm (Java, C++, C#, ...)
- Python supports OO programming, and all things in Python are objects (see also slides 35 pp)

Environments and Python Package Index

Why virtual environments?

- install multiple versions of the same library (in different environments)
- good practice (reproducibility, managing different projects)

Given an installed Python interpreter, we can create virtual environments:

```
python -m venv myvirtualenv
```

We need to activate them:

```
source myvirtualenv/bin/activate
```

- The Python Package Index (PyPI) provides many python packages (<https://pypi.org>)
- Can search the website for packages, and available versions
- Install locally (in virtual environment) using `pip`

Example: install the python cowsay package:

```
pip install cowsay
```

Uninstall:

```
pip install cowsay
```


- `pip install cowsay`
- `pip install cowsay==3.0`
 - install version 3.0
- `pip uninstall cowsay`
- `pip install -U cowsay`
 - upgrade cowsay
- `pip show cowsay`
 - show information about installed package
- `pip list`
 - list installed packages
- `pip freeze`
 - list installed packages in machine readable format

Summary virtual environments and pip commands

Summary

- create virtual environment before installing packages
- Common names for virtual environments: `env`, `venv`, `.env`, `.venv`
- use (at least) one virtual environment per project

- use

```
pip freeze
```

and

```
pip install -r requirements.txt
```

to maintain reproducible environments

See more detailed discussion at: <https://fangohr.github.io/introduction-to-python-for-computational-science-and-engineering/18-environments.html>

*For Anaconda users: interaction conda and pip

Anaconda provides packages and (conda) environments through conda.

- Avoid mixing pip installs with conda installs, i.e.
 - if conda can install all the required packages, then use that
 - if conda cannot install the required package, either
 - first install all that is needed/available from conda
 - then install the desired packages through pip that conda cannot provide
 - afterwards, do not use conda again to install more packages.
- or (if possible)
- install all packages from pip

See also <https://www.anaconda.com/blog/using-pip-in-a-conda-environment>

ODEs

Ordinary Differential Equations

- Many processes, in particular *time-dependent* processes, can be described as Ordinary Differential Equations (ODEs), such as dynamics of engineering systems, quantum physics, chemical reactions, biological systems modelling, and population dynamics.
- ODEs have *exactly one* independent variable t (often, but not always representing time).
- The simplest ODE type has one degree of freedom, y , which depends on the time t , i.e. $y = y(t)$. (For example temperature as a function of time, the distance a car has moved as function of time, etc.)

- In general, a vector \mathbf{y} with k components can depend on the independent variable t : this is a *system* of ordinary differential equations with k degrees of freedom.
- The *solution* of the ODE is the function $y(t)$.
- We are typically being given
 - an initial value y_0 of $y(t)$ at some time t_0 and
 - the ODE itself which relates the change of y with t to some function $f(t, y)$, i.e.

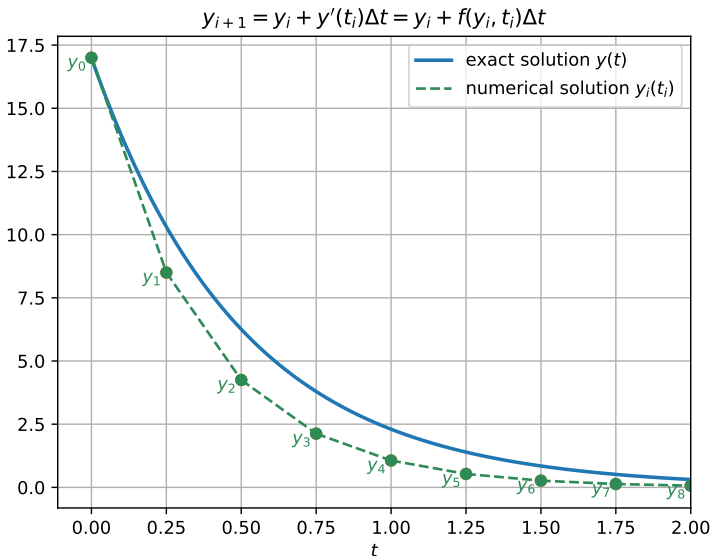
$$\frac{dy}{dt} = f(t, y) \quad (14)$$

- Example: looking for solution $y(t)$ from $t_0 = 0$ to $t = 2$ of

$$\frac{dy}{dt} = -2y \quad \text{with} \quad y_0 = y(t_0) = 17$$

The exact solution is $y(t) = 17 \exp(-2t)$.

Principle of finding numerical solution to ODE



Interface solve_ivp

- aim: solve

$$\frac{dy}{dt} = f(t, y)$$

- `from scipy.integrate import solve_ivp`
- `solve_ivp` has the following input and output parameters:
`sol = solve_ivp(f, t_span, y0)`

Input:

- `f` is function `f(t, y)` that returns the right-hand side
- `t_span` is a tuple `(t0, tf)` describing the span of `t` for which we search the solution
- `y0` is the initial value of the solution at time `t0` (i.e. `y0 = y(t0)`)

Output:

- `sol` is a `OdeResult` object that contains the solution

Using solve_ivp – example 1

Require solution $y(t)$ from $t = 0$ to $t = 2$ of

$$\frac{dy}{dt} = -2y \quad \text{with} \quad y(0) = 17$$

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

def f(t, y):
    """this is the rhs of the ODE to integrate, i.e. dy/dt=f(y,t)"""
    return -2 * y

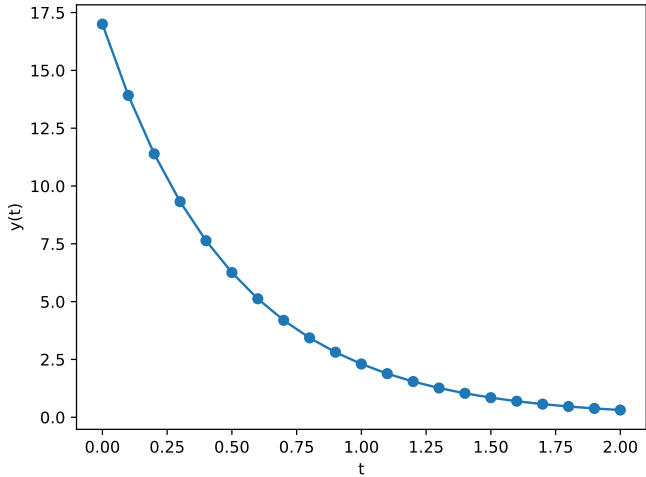
y0 = [17] # initial value y0=y(t0)
t0 = 0 # integration limits for t: start at t=0
tf = 2 # and finish at t=2
t_eval = np.linspace(t0, tf, 21)

sol = solve_ivp(fun=f, t_span=[t0, tf], y0=y0, t_eval=t_eval)

fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0], "o-"); ax.set_xlabel("t");
```

Using solve_ivp – example 1, solution

Solution:



Using solve_ivp – example 2

Require solution $y(t)$ from $t = 0$ to $t = 2$ of

$$\frac{dy}{dt} = -\frac{1}{100}y + \sin(10\pi t) \quad \text{with} \quad y(0) = -2$$

```
import math
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

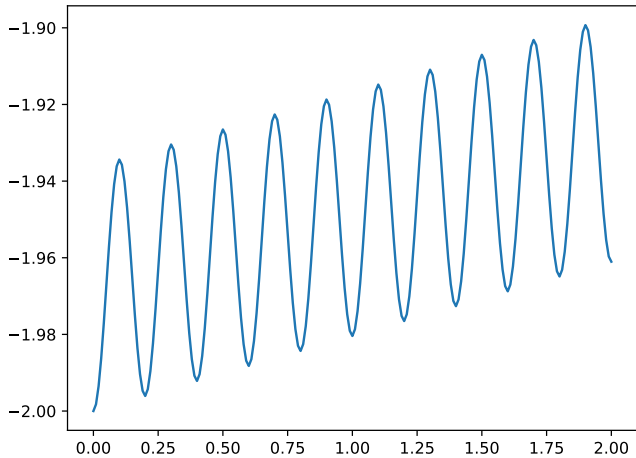
def f(t, y):
    return -0.01 * y + math.sin(10 * math.pi * t)

ts = np.arange(0, 2.01, 0.01)
y0 = [-2]
sol = solve_ivp(f, (0, 2), y0, t_eval=ts, atol=1e-8, rtol=1e-8)

fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0])
ax.set_xlabel("t"); ax.set_ylabel("y(t)")
fig.savefig("odeintexample2.pdf")
```

Using solve_ivp – example 2, solution

Solution:



2nd order ODE

- Any second order ODE can be re-written as two coupled first order ODE
- Example: Harmonic Oscillator (HO)
 - Differential equation $\frac{d^2r}{dt^2} = -\omega^2r$ or short $r'' = -\omega^2r$
 - Introduce $v = r'$
 - rewrite equation as two first order equations

$$r'' = -\omega^2r \quad \longrightarrow \quad \begin{aligned} v' &= -\omega^2r \\ r' &= v \end{aligned}$$

- General strategy:
 - convert higher order ODE into a set of (coupled) first order ODE
 - use computer to solve set of 1st order ODEs

2nd order ODE – using `solve_ivp`

- One 2nd order ODE \rightarrow 2 coupled 1st order ODEs
- Integration of *system* of 1st order ODEs:
 - “pretty much like integrating one 1st order ODE” but
 - y is now a vector (and so is f):

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \iff \begin{pmatrix} \frac{dy_1}{dt} \\ \frac{dy_2}{dt} \end{pmatrix} = \begin{pmatrix} f_1(t, \mathbf{y}) \\ f_2(t, \mathbf{y}) \end{pmatrix}$$

- need to pack and unpack variables into the *state vector* \mathbf{y} :
- Example harmonic oscillator:
 - decide to use this packing: $\mathbf{y} = (r, v)$
 - then \mathbf{f} needs to return $\mathbf{f} = \left(\frac{dr}{dt}, \frac{dv}{dt}\right)$
- the `sol` object returned by `solve_ivp` has an attribute `sol.y` which contains a vector \mathbf{y} for every time step
 - need to extract results for r and v from that matrix \rightarrow see next slide

2nd order ODE – Python solution harmonic oscillator (HO)

```
from numpy import array, arange
from scipy.integrate import solve_ivp

def f(t, y):
    # right hand side, takes array(!) y
    omega = 1
    r = y[0] # extract r from array y
    v = y[1] # extract v from array y
    drdt = v # compute right hand side
    dvdt = -omega ** 2 * r
    return array([drdt, dvdt]) # return array

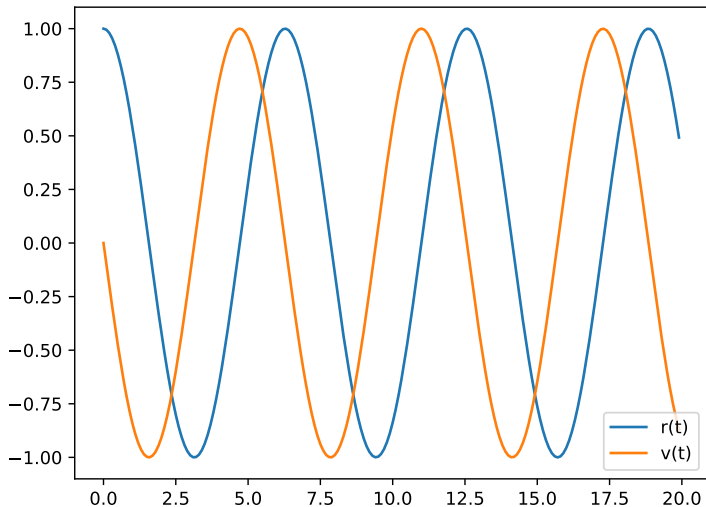
ts = arange(0, 20, 0.1) # required times for solution
r0 = 1 # initial r
v0 = 0 # initial v
y0 = [r0, v0] # combine r and v into y

sol = solve_ivp(f, (0, 20), y0, t_eval=ts) # solve ODEs

rs = sol.y[0] # extract result: r(t)
vs = sol.y[1] # extract result: v(t)
```

2nd order ODE – result

Solution



Summary 2nd order system

- Strategy:
 - transform one 2nd order ODE into 2 (coupled) first order ODEs
 - solve both first order ODEs simultaneously
- Need to use vectors (typically “arrays”) in to pass state vector to right-hand-side function.
- Use example on previous slides as guidance.

2 Coupled ODEs: Predator-Prey problem

- Predator and prey. Let
 - $p_1(t)$ be the number of rabbits
 - $p_2(t)$ be the number of foxes
- Time dependence of p_1 and p_2 :
 - Assume that rabbits proliferate at a rate a . Per unit time a number ap_1 of rabbits is born.
 - Number of rabbits is reduced by collisions with foxes. Per unit time cp_1p_2 rabbits are eaten.
 - Assume that birth rate of foxes depends only on food intake in form of rabbits.
 - Assume that foxes die a natural death at a rate b .
- Numbers
 - rabbit birth rate $a = 0.7$

- rabbit-fox-collision rate $c = 0.007$
- fox death rate $b = 1$
- Put all together in predator-prey ODEs

$$p_1' = ap_1 - cp_1p_2$$

$$p_2' = cp_1p_2 - bp_2$$

- Solve for $p_1(0) = 70$ and $p_2(0) = 50$ for 30 units of time:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

def rhs(t, y):
    a = 0.7
    c = 0.007
    b = 1
    p1 = y[0]
    p2 = y[1]

    dp1dt = a * p1 - c * p1 * p2
    dp2dt = c * p1 * p2 - b * p2

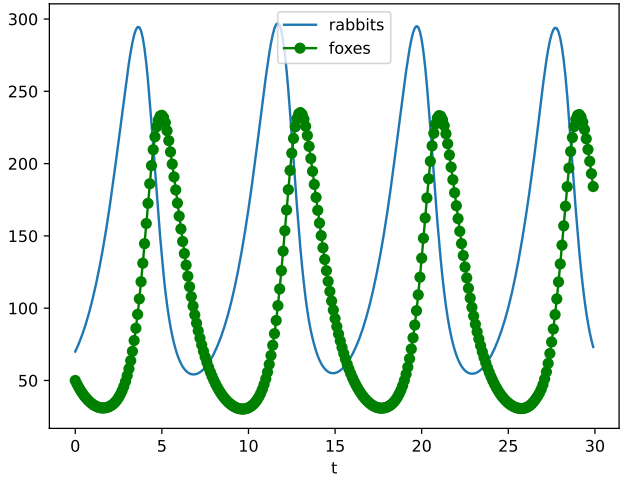
    return np.array([dp1dt, dp2dt])

p0 = [70, 50]      # initial condition
t0 = 0
tfinal = 30
ts = np.arange(t0, tfinal, 0.1)
```

```
sol = solve_ivp(rhs, [t0, tfinal], p0, t_eval=ts)

p1 = sol.y[0]           # extract p1 and
p2 = sol.y[1]           # p2

fig, ax = plt.subplots()
ax.plot(sol.t, p1, label='rabbits')
ax.plot(sol.t, p2, '-og', label='foxes')
ax.legend()
ax.set_xlabel('t')
fig.savefig('predprey.pdf')
```



Outlook

Suppose we want to solve a (vector) ODE based on Newton's equation of motion in three dimensions:

$$\frac{d^2\mathbf{r}}{dt^2} = \frac{\mathbf{F}(\mathbf{r}, \mathbf{v}, t)}{m}$$

Rewrite as two first order (vector) ODEs:

$$\begin{aligned}\frac{d\mathbf{v}}{dt} &= \frac{\mathbf{F}(\mathbf{r}, \mathbf{v}, t)}{m} \\ \frac{d\mathbf{r}}{dt} &= \mathbf{v}\end{aligned}$$

Need to pack 6 variables into "y": for example

$$\mathbf{y} = (r_x, r_y, r_z, v_x, v_y, v_z)$$

Right-hand-side function $\mathbf{f}(\mathbf{y}, t)$ needs to return:

$$\mathbf{f} = \left(\frac{dr_x}{dt}, \frac{dr_y}{dt}, \frac{dr_z}{dt}, \frac{dv_x}{dt}, \frac{dv_y}{dt}, \frac{dv_z}{dt} \right) \quad (15)$$

Outlook examples

- Example: Molecular dynamics simulations have one set of 6 degrees of freedom as in equation (15) *for every atom* in their simulations.
- Example: Material simulations discretise space into finite elements, and for dynamic simulations the number of degrees of freedom are proportional to the number of nodes in the mesh.
- Very sophisticated time integration schemes for ODEs available (such as "sundials" suite).
- The tools in `scipy.integrate` are pretty useful already (`solve_ivp` has multiple solvers - we have only used the default Runge Kutta 45 solver.).

Sympy

What?

- symbolic algebra - computing with variables not numbers (like Mathematica, SageMath, Wolfram Alpha, other, ...)

Why?

- Use symbolic computation before moving to numerical calculations to avoid mistakes
- and to simplify expression as much as possible.
- Write computer code (or LaTeX) automatically from sympy
- Or use from Python using `sympy.lambdify`

Why symbolic python?

- sympy is not the only option - other packages may well be faster/know more mathematics, but
 - sympy connects well to Python ecosystem of computational science tools
 - free and open source
 - scriptable: can integrate into *automatic* workflows
 - very powerful

Symbolic Python - basics

```
>>> import sympy
>>> x = sympy.Symbol('x') # define symbolic
>>> y = sympy.Symbol('y') # variables
>>> x + x
2*x
>>> t = (x + y)**2
>>> print(t)
(x + y)**2
>>> sympy.expand(t)
x**2 + 2*x*y + y**2
>>> sympy.pprint(t) # PrettyPRINT
      2
(x + y)
>>> sympy.printing.latex(t) # Latex export
'\\left(x + y\\right)^{2}'
```

Substituting values and numerical evaluation

```
>>> t
(x + y)**2
>>> t.subs(x, 3)           # substituting variables
(y + 3)**2                 # or values
>>> t.subs(x, 3).subs(y, 1)
16
>>> n = t.subs(x, 3).subs(y, sympy.pi)
>>> print(n)
(3 + pi)**2
>>> n.evalf()              # EVALuate to Float
37.7191603226281
>>> p = sympy.pi
>>> p
pi
>>> p.evalf()
```

```
3.14159265358979
```

```
>>> p.evalf(47) # request 47 digits
```

```
3.1415926535897932384626433832795028841971693993
```

Working with infinity

```
>>> from sympy import limit, sin, oo
>>> limit(1/x, x, 50)          # what is 1/x if x --> 50
1/50
>>> limit(1/x, x, oo)         # oo is infinity
0
>>> limit(sin(x) / x, x, 0)
1
>>> limit(sin(x)**2 / x, x, 0)
0
>>> limit(sin(x) / x**2, x, 0)
oo
```

Taylor series

```
>>> from sympy import series
>>> taylorseries = series(sin(x), x, 0)
>>> taylorseries
x - x**3/6 + x**5/120 + O(x**6)
>>> sympy.pprint(taylorseries)
      3      5
      x      x
x - -- + --- + O(x**6)
   6    120
>>> taylorseries = series(sin(x), x, 0, n=10)
>>> sympy.pprint(taylorseries)
      3      5      7      9
      x      x      x      x
x - -- + --- - ---- + ----- + O(x**10)
   6    120  5040  362880
```



```
>>> from sympy import integrate
>>> a, b = sympy.symbols('a, b')
>>> integrate(2*x, (x, a, b))
-a**2 + b**2
>>> integrate(2*x, (x, 0.1, b))
b**2 - 0.01
>>> integrate(2*x, (x, 0.1, 2))
3.9900000000000000
```

Solving equations

Finally, we can solve non-linear equations, for example:

```
>>> (x + 2)*(x - 3)      # define quadratic equation
                                # with roots x=-2, x=3

(x - 3)*(x + 2)
>>> r = (x + 2)*(x - 3)
>>> r.expand()
x**2 - x - 6
>>> sympy.solve(r, x)    # solve r = 0
[-2, 3]                  # solution is x = -2, 3
```

Lambdify sympy expressions

```
>>> from sympy import sin, cos, symbols, lambdify
>>> import numpy as np
>>> x = symbols('x')
>>> symb = sin(x) + cos(x)
>>> symb
sin(x) + cos(x)
>>> f = lambdify(x, symb, 'numpy')
>>> f(0)
1.0
>>> f(np.linspace(0, 1, 10))
array([1.          , 1.10471614, 1.19580783, 1.27215164,
1.33280603, 1.37702295, 1.40425706, 1.4141725 ,
1.40664697, 1.38177329])
```

Workflow: Create sympy expressions, then lambdify them to execute faster.

Sympy summary

- Sympy is purely Python based
- fairly powerful (although better open source tools are available if required)
- we should use computers for symbolic calculations routinely alongside pen and paper, and numerical calculations
- can produce \LaTeX output
- can produce C and fortran code (and wrap this up as a python function automatically (“autowrap”))

Testing

- Writing software is easy – debugging it is hard
- When debugging, we always *test*
- Later code changes may require repeated testing
- Best to *automate testing* by writing functions that contain tests
- A big topic: here we provide some key ideas
- We use Python extension tool `py.test`, see pytest.org

Example 1: mixstrings.py

```
def mixstrings(s1, s2):  
    """Given two strings s1 and s2, create and return a new  
    string that contains the letters from s1 and s2 mixed:  
    i.e. s[0] = s1[0], s[1] = s2[0], s[2] = s1[1],  
    s[3] = s2[1], s[4] = s1[2], ...  
    If one string is longer than the other, the extra  
    characters in the longer string are ignored.
```

Example:

```
>>> mixstrings("Hello", "12345")  
'H1e2l3l4o5'  
""  
# what length to process  
n = min(len(s1), len(s2))  
# collect chars in this list  
s = []
```



```
for i in range(n):
    s.append(s1[i])
    s.append(s2[i])
return "".join(s)

def test_mixstrings_basics():
    assert mixstrings("hello", "world") == "hweolrllod"
    assert mixstrings("cat", "dog") == "cdaotg"

def test_mixstrings_empty():
    assert mixstrings("", "") == ""

def test_mixstrings_different_length():
    assert mixstrings("12345", "123") == "112233"
    assert mixstrings("", "hello") == ""

if __name__ == "__main__":
    test_mixstrings_basics()
    test_mixstrings_empty()
    test_mixstrings_different_length()
```

- tests are run if `mixstrings.py` is the top-level (tests are not run if file is imported)
- no output if all tests pass (“*no news is good news*”)
- More common approach than calling tests from `__main__`: use `py.test`.

py.test (also known as pytest)

We can use the standalone program `py.test` to run test functions in *any* python program:

- `py.test` will look for functions with names starting with `test_`
- and execute each of those as one test.
- Example:

```
$> py.test -v mixstrings.py
===== test session starts =====
platform darwin -- Python 3.10.2, pytest-7.1.2
collected 3 items

mixstrings.py::test_mixstrings_basics PASSED [ 33%]
mixstrings.py::test_mixstrings_empty PASSED [ 66%]
mixstrings.py::test_mixstrings_different_length PASSED [100%]
===== 3 passed in 0.01s =====
```

- This works, even if the file to be tested (here `mixstrings.py`) does not refer to `pytest` at all.

*Calling pytest from a python file

If desired, one can trigger execution of `pytest` from python file.

Example:

```
import pytest
```

<parts of the file missing here>

```
if __name__ == "__main__":  
    pytest.main(["-v", "mixstrings.py"])
```

However, it is much more common to use `py.test` to discover and execute the tests (often across multiple files).

Advanced Example 3: factorial.py

For reference: In this example, we check that an exception is raised if a particular error is made in calling the function.

```
import math
import pytest

def factorial(n):
    """ Compute and return n! recursively.
    Raise ValueError if n is negative or non-integer.

    >>> from myfactorial import factorial
    >>> [factorial(n) for n in range(5)]
    [1, 1, 2, 6, 24]
    """

    if n < 0:
        raise ValueError(f"n should be > 0 but n={n}")
```

```
if isinstance(n, int):
    pass
else:
    raise TypeError(f"n must be integer but is {type(n)}.")

# actual calculation
if n == 0:
    return 1
else:
    return n * factorial(n - 1)

def test_basics():
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(3) == 6

def test_against_standard_lib():
    for i in range(20):
        assert math.factorial(i) == factorial(i)

def test_negative_number_raises_error():
```

```

with pytest.raises(ValueError):
    factorial(-1)
# this will pass if
# factorial(-1) raises
# a ValueError

def test_noninteger_number_raises_error():
    with pytest.raises(TypeError):
        factorial(0.5)

```

Output from successful testing:

```

$> py.test -v factorial.py
===== test session starts =====
platform darwin -- Python 3.10.2, pytest-7.1.2
collected 4 items

factorial.py::test_basics PASSED [ 25%]
factorial.py::test_against_standard_lib PASSED [ 50%]
factorial.py::test_negative_number_raises_error PASSED [ 75%]
factorial.py::test_noninteger_number_raises_error PASSED [100%]
===== 4 passed in 0.02s =====

```

Notes on pytest

- Normally, we call `py.test` from the command line
- Either give filenames to process (will look for functions starting with `test` in those files)
- or let `py.test` autodiscover all files (!) starting with `test` to be processed.

Example:

```
===== test session starts =====
platform darwin -- Python 3.10.2, pytest-7.1.2
collected 7 items

mixstrings.py::test_mixstrings_basics PASSED [ 14%]
mixstrings.py::test_mixstrings_empty PASSED [ 28%]
mixstrings.py::test_mixstrings_different_length PASSED [ 42%]
factorial.py::test_basics PASSED [ 57%]
factorial.py::test_against_standard_lib PASSED [ 71%]
factorial.py::test_negative_number_raises_error PASSED [ 85%]
factorial.py::test_noninteger_number_raises_error PASSED [100%]
===== 7 passed in 0.01s ===== 399
```


Testing summary

- Unit testing, integration testing, regression testing, system testing
- absolute key role in modern software engineering: always write (some) tests for your software
- bigger projects have "continuous integration testing": automatic execution of tests on any change
- "eXtreme Programming" (XP) philosophy suggests to write tests *before* you write code ("test-driven-development (TDD)")

Executable `py.test` and python module `pytest` are not part of the standard python library.

Object Oriented Programming

- Motivation and terminology
- Time example
 - encapsulation
 - defined interfaces to hide data and implementation
 - operator overloading
 - inheritance
 - (teaching example only: normally `datetime` and others)
- Geometry example
- Objects we have used already
- Summary

Motivation

- When programming we often store *data*
- and *do* something with the data.
- For example,
 - an array keeps the data and
 - a function does something with it.
- Programming driven by actions (*i.e.* calling functions to do things) is called *imperative* or *procedural* programming.

Object Orientation

- merge data and functions (that operate on this data) together into *classes*.

(...and objects are “instances of a class”)

- a class combines data and functions
(think of a class as a blue print for an object)
- objects are *instances* of a class
(you can build several objects from the same blue print)
- a class contains *members*
- members of classes that store data are called *attributes*
- members of classes that are functions are called *methods*
(or behaviours)

Example 1: a class to deal with time

```
class Time:
    def __init__(self, hour, min):
        self.hour = hour
        self.min = min

    def print24h(self):
        print(f"{self.hour:2}:{self.min:2}")

    def print12h(self):
        if self.hour < 12:
            ampm = "am"
        else:
            ampm = "pm"

        print(f"{self.hour % 12:2}:{self.min:2} {ampm}")

if __name__ == "__main__":
```

```
t = Time(15, 45)

print("print as 24h: "),
t.print24h()
print("print as 12h: "),
t.print12h()

print(f"The time is {t.hour} hours and {t.min} minutes.")
```

which produces this output:

```
print as 24h:
15:45
print as 12h:
 3:45 pm
The time is 15 hours and 45 minutes.
```

- `class Time`: starts the definition of a class with name `Time`
- `__init__` is the *constructor* and is called whenever a new object is initialised
- all methods in a class need `self` as the first argument. `Self` represents the object.
- variables can be stored and are available everywhere *within* the object when assigned to `self`, such as `self.hour` in the example.
- in the main program:
 - `t = Time(15, 45)` creates the object `t`
↔ `t` is an instance of the class `Time`
 - methods of `t` can be called like this `t.print24h()`.

This was a mini-example demonstrating how data attributes (*i.e.* `hour` and `min`) and methods (*i.e.* `print24h()` and `print12h()`) are combined in the `Time` class.

Members of an object

- In Python, we can use `dir(t)` to see the members of an object `t`. For example:

```
>>> t = Time(15, 45)
>>> dir(t)
['__class__', '__doc__', ...<entries removed here>....,
 'hour', 'min', 'print12h', 'print24h']
```

- We can also modify attributes of an object using for example `t.hour = 10`. However, *direct* access to attributes is sometimes suppressed (although it may look like direct access → property).

Data hiding (also: information hiding)

- A well designed class provides methods to get and set attributes.
- These methods define the *interface* to that class.
- Purpose of get and set methods:
 - to perform error and consistency checking when values are set
 - to hide the implementation of the class (from the user):
 - we can change the implementation of the class without changing the interface (and a user of the class would never know)
 - makes future changes possible
- We introduce set and get methods as one would use in Java and C++ to reflect the common ground in OO class design. *In Python, the use of **property** is often preferred over set and get methods.*

Example 2: a class to deal with time

```
class Time:
    def __init__(self, hour, min):
        self.setHour(hour)
        self.setMin(min)

    def setHour(self, hour):
        if 0 <= hour <= 23:
            self._hour = hour
        else:
            raise ValueError(f"Invalid hour value: {hour}")

    def setMin(self, min):
        if 0 <= min <= 59:
            self._min = min
        else:
            raise ValueError(f"Invalid min value: {min}")
```

```
def getHour(self):
    return self._hour

def getMin(self):
    return self._min

def print24h(self):
    print(f"{self.getHour():2}:{self.getMin():02}")

def print12h(self):
    if self._hour < 12:
        ampm = "am"
    else:
        ampm = "pm"

    print(f"{self._hour%12:2}:{self._min:2} {ampm}")

if __name__ == "__main__":
    t = Time(15, 45)
```

```
print("print as 24h: "),
t.print24h()
print("print as 12h: "),
t.print12h()
print(f"that is {t.getHour()} hours and {t.getMin()} minutes")
```

which produces

```
print as 24h:
15:45
print as 12h:
 3:45 pm
that is 15 hours and 45 minutes
```

- providing *set* and *get* methods for attributes of an object
- The pythonic way for get and set functions is through *properties*. A property is a special attribute:
 - get and set functions are called automatically when the attribute is accessed or assigned to.
- Advanced: Attributes that the user cannot access directly are called *private*.
 - In Python, class attributes can never be truly private. (in contrast to C++, Java, ...)
 - Convention: an attribute starting with an underscore is private, and should not be accessed directly (by the user of the class). Example: `self._hour`

Operator overloading

- We constantly use operators to “do stuff” with objects.
- What the operator does, depends on the objects it operates on. For example:

```
>>> a = "Hello "; b = "World"
>>> a + b                                # concatenation
'Hello World'
>>> c = 10; d = 20
>>> c + d                                # addition
30
```

- This is called *operator overloading* because the operation is overloaded with more than one meaning.
- Other operators include -, * , **, [], (), >, >=, ==, <=, <, str(), repr(), ...
- We can overload these operators for our own objects. The next slide shows an example that overloads the > operator for the Time class.
- It also overloads the “str” and “repr” functions.

```
class Time:
    def __init__(self, hour, min):
        self.hour, self.min = hour, min

    def __str__(self):
        """overloading the str operator (STRing)"""
        return f"[ {self.hour:2d}:{self.min:2d} ]"

    def __repr__(self):
        """overloading the repr operator (REPResentation)"""
        return f"Time({self.hour:2d}, {self.min:2d})"

    def __gt__(self, other):
        """overloading the GreaterThan operator"""
        selfminutes = self.hour * 60 + self.min
        otherminutes = other.hour * 60 + other.min
        return selfminutes > otherminutes

if __name__ == "__main__":
    t1 = Time(15, 45)
```



```
t2 = Time(10, 55)

print(f"Informal string representation of t1: {str(t1)}")
print(f"Representation of object = {repr(t1)}")

print("compare t1 and t2: "),
if t1 > t2:
    print("t1 is greater than t2")
```

Output:

```
Informal string representation of t1: [ 15:45 ]
Representation of object = Time(15, 45)
compare t1 and t2:
t1 is greater than t2
```

Inheritance

- Sometimes, we need classes that share certain (or very many, or all) attributes but are slightly different.
- Example 1: Geometry
 - a point (in 2 dimensions) has an x and y attribute
 - a circle is a point with a radius
 - a cylinder is a circle with a height
- Example 2: People at universities
 - A person has an address.
 - A student is a person and selects modules.
 - A lecturer is a person with teaching duties.
 - ...
- In these cases, we define a *base class* (or *parent class*) and *derive* other classes from it.
- This is called *inheritance*

Inheritance example Time

```
class Time:
    def __init__(self, hour, min):
        self.hour = hour
        self.min = min

    def __str__(self):
        """overloading the str operator (STRING)"""
        return f"[ {self.hour:2}:{self.min:02} ]"

    def __gt__(self, other):
        """overloading the GreaterThan operator"""
        selfminutes = self.hour * 60 + self.min
        otherminutes = other.hour * 60 + other.min
        return selfminutes > otherminutes
```

```
class TimeUK(Time):
    """Derived (or inherited class)"""
    def __str__(self):
        """overloading the str operator (STRing)"""
        if self.hour < 12:
            ampm = "am"
        else:
            ampm = "pm"

        return f"[ {self.hour%12:2}:{self.min:02} {ampm}]"

if __name__ == "__main__":
    t3 = TimeUK(15, 45)
    t4 = Time(16, 15)
    print(t3, t4)

    if t3 > t4:
```

```
    print("t3 is greater than t4")
else:
    print("t3 is not greater than t4")
```

Output:

```
[ 3:45 pm] [ 16:15 ]
t3 is not greater than t4
```

*Inheritance example Geometry

```
import math

class Point:                                # this is the base class
    """Class that represents a point """
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

class Circle(Point):                        # is derived from Point
    """Class that represents a circle """
    def __init__(self, x=0, y=0, radius=0):
        Point.__init__(self, x, y)
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2
```

```

class Cylinder(Circle):          # is derived from Circle
    """Class that represents a cylinder"""

    def __init__(self, x=0, y=0, radius=0, height=0):
        Circle.__init__(self, x, y, radius)
        self.height = height

    def volume(self):
        return self.area() * self.height

if __name__ == "__main__":
    d = Circle(x=0, y=0, radius=1)
    print("circle area:", d.area())
    print("attributes of circle object are")
    print([name for name in dir(d) if name[:2] != "__"])
    c = Cylinder(x=0, y=0, radius=1, height=2)
    print("cylinder volume:", c.volume())
    print("attributes of cylinder object are")
    print([name for name in dir(c) if name[:2] != "__"])

```

Output:

```
circle area: 3.141592653589793
attributes of circle object are
['area', 'radius', 'x', 'y']
cylinder volume: 6.283185307179586
attributes of cylinder object are
['area', 'height', 'radius', 'volume', 'x', 'y']
```


*Inheritance (2)

- if class A should be derived from class B we need to use this syntax:
`class A(B):`
- Can call constructor of base class explicitly if necessary (such as in `Circle` calling of `Point.__init__(...)`)
- Derived classes inherit attributes and methods from base class (see output on previous slide: for example the `cylinder` and `circle` object have inherited `x` and `y` from the `point` class).

*super()

In the Circle class definition, we can replace

```
Point.__init__(self, x, y)
```

with

```
super().__init__(x, y)
```

as a short cut to call a method from the (single) parent class.

(Same for the Cylinder class definition.)

Everything in Python is an object

All “things” in Python are objects, including numbers, strings and functions.

```
>>> dir(42)           # numbers are objects
>>> dir(list)         # list is an object
>>> import math
>>> dir(math)         # modules are objects
>>> dir(lambda x: x)  # functions are objects
```

Summary

- Object orientation is about merging data and functions into one object (sometimes called encapsulation).
- Data hiding (through get and set methods) makes the classes more flexible: easier to maintain, possible to change internal implementation
- Through operator overloading we can make working with the objects more convenient and more flexible
- Classes can be derived from other classes: facilitates re-use of code

Pandas

- de-facto standard in data science (and machine learning)
- builds on numpy
- convenient handling of multi-dimensional data sets
- important data structures: **Series** and **DataFrame**
- excellent import and export functionality, including **csv** and **xlsx**.
- many, many, many parameters, functions, tools (Can't know them all)
- for data cleaning and data exploration typically used in Jupyter Notebook

See <https://fangohr.github.io/introduction-to-python-for-computational-science-and-engineering/17-pandas.html>

Practical computational science recommendations

- use version control
- start in Python
- use tests
- keep it simple
- make it readable
- use notebooks for examples -> documentation (sphinx)
- if you need to change/extend/rewrite software
 - automatic tests are your friend
 - continuous integration
 - we can also *test documentation* (nbval, doctest)

Software too slow?

- Identify where it is slow ("Profiling")
- move execution of 'slow' operations to compiled code where necessary
 - through use of numpy
 - through use of Cython
 - through linking to compiled code (ctypes, cython, swig, boost, f2py, ...)
- Parallelise through use of libraries that can execute in parallel
 - mkl numpy
 - dask
 - numba
 - pytorch, cupy, ...

Includes use of GPUs.

- use notebooks to document computational work
- use version control (for software, reports and papers)
- archive software and notebooks (publicly if you can)
- in particular for (more reproducible) publications [1]:
 - publish git repo with paper (Zenodo?)
 - document your software environment
 - if you can create it automatically, this is best
 - consider making your repo **binder**-enabled ()

[1] Beg, Fangohr, etal: *Using Jupyter for reproducible scientific workflows*, Computing in Science and Engineering 23, 36-46 [10.1109/MCSE.2021.3052101](https://doi.org/10.1109/MCSE.2021.3052101) (2021)

Give back to the community where you can

- Contribute to the open source tools you are using, for example
 - provide bug reports
 - suggest improvements to documentation
 - make feature requests
 - helping other users
 - ...
- Cite software that is important for your work in your papers: many packages suggest what to cite if you use them

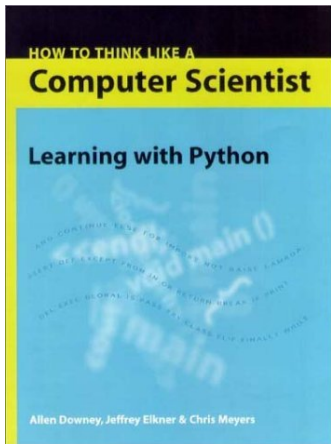
What to learn next?

What programming language to learn how - suggestions

- ...it all depends what you want to achieve:
- To learn C or Fortran, get a book or on-line tutorial.
- To learn object oriented programming (OOP), read a more detailed introduction, for example “How to think like a computer Scientist” (for Python)
- To learn C++, learn OOP using Python first, then switch to C++.

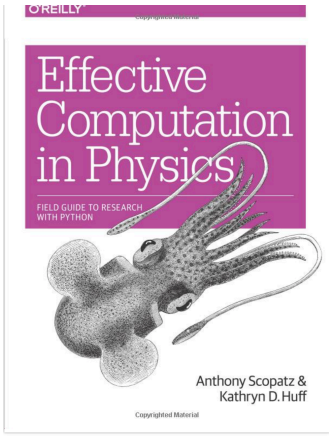
Note:

- Python provides an excellent platform for all possible tasks
⇒ it could well be all you need for some time to come.



- “How to Think Like a Computer Scientist: Learning with Python”. (ISBN 0971677506) free at <http://greenteapress.com/wp/think-python-2e/>
- Very systematic presentation of all important programming concepts, including OOP.
- Aimed at first year computer science students.
- Recommended if you like programming and want to know more about the concepts.

Further reading (2)



- “Effective Computation in Physics” (ISBN 1491901535, 2015)
- Good overview of Python language and Python-based computational ecosystem
- Also introduces data processing, version control (Git), Github, testing, HDF5, parallelism, software deployment

What other tools, methods and topics might be relevant?

Tools to extend your computational toolkit / suggested self-study topics

- Systematic testing (py.test for python) and
- Test Driven Development (TDD)
- Version control (Git)
- Automate everything (continuous integration)
- Jupyter Notebook
- \LaTeX (professional document creation)
- Editor (Emacs? Vim? Sublime? Eclipse? ...)
- Containers (Docker / Singularity / ...)
- Data management: hdf5
- Performance: parallel computing
- Touch typing
- ...

Typing

Dynamic Typing

Python derives flexibility from being dynamically typed:

```
def add(x, y):  
    """Type of x and y is dynamic."""  
    print(f"Type of {x=} is {type(x)}")  
    return x + y  
  
print(add(10, 20))  
print(add("Hello", " World"))
```

Output:

```
Type of x=10 is <class 'int'>  
30  
Type of x='Hello' is <class 'str'>  
Hello World
```

Duck typing — behaviour more important than type

```
def print_length(x):  
    """Works for every object with __len__ method."""  
    print(f"The object of type {type(x)} has length {len(x)}.")
```

```
class Len42class:  
    """A class where every object has length 42."""  
    def __len__(self):  
        return 42
```

```
x = [10, 20]  
print_length(x) # list has length  
y = Len42class() # y has length  
print_length(y)
```

Output:

```
The object of type <class 'list'> has length 2.  
The object of type <class '__main__.Len42class'> has length 42.
```

- More formal “static typing” information can be useful:
 - better (machine readable) documentation of types
 - static type checking may discover mistakes
 - editors/IDEs can use static type information
 - potential execution speed-up (see cython)
- **Typing module** for type annotation introduced in Python 3.5
- Relevant PEPs: [PEP483](#) and [PEP484](#)
- More concise introduction to typing [realpython.com](#)

Static typing example

- Function type annotation: expect `str` and return `str`

```
1 def hello(name: str) -> str:
2     """Given a name, return 'Hello ' + name."""
3     return "Hello " + name
4
5 hello("Paul") # correct function call
6 hello(42) # incorrect type
```

- Can use `mypy` to do static type analysis:

```
typing-static1.py:6: error: Argument 1 to "hello"
↳ has incompatible type "int"; expected "str"
Found 1 error in 1 file (checked 1 source file)
```

- *gradual* introduction of type annotations is possible: can introduce type annotation for some functions only
- effective to annotate most heavily used functions first
 - they are called from other places
 - accidental calls with incorrect types can be discovered

Gradual typing example

```
1 • def mysum(a: int, b: int) -> int:
2     """Expect two ints and return the sum."""
3     return a + b
4
5 def f_without_types(x):
6     """Return x. A function without type annotation."""
7     return x
8
9 print(mysum(2, 3))
10 print(mysum("Hello", 2023)) # will not work
```

- Can use mypy to do static type analysis:

```
typing-gradual.py:10: error: Argument 1 to "mysum"
↳ has incompatible type "str"; expected "int"
Found 1 error in 1 file (checked 1 source file)
```


Useful tools



“one style, as long as it is this one”

- leave formatting to **black**
- focus on content (rather than formatting)
- makes code review easier
- compatible with PEP8

Usage:

- Check if **file.py** sticks to Black standard:
black --check file.py
- Autoformat **file.py**:
black file.py
- Can be used by editors (e.g. Spyder) and tools (e.g. pre-commit)

Code with type annotations (see slide 439) can be analysed statically (i.e. without being executed).

Important tools:

- mypy <https://mypy-lang.org/>
- pytype <https://github.com/google/pytype>

Pytype can also infer types (to some degree) and merge to source.

A gentle introduction to the topic in [Talk Python to Me podcast, episode 151](#).

commit 88dcec6a44a667436647f28a85c3e62b806eace1
Author: Hans Fangohr <hans.fangohr@mpsd.mpg.de>
Date: Fri Jan 27 14:28:17 2023 +0100

minted font size adjustments